RICE UNIVERSITY

A Framework for Testing Concurrent Programs

by

Mathias Guenter Ricken

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Robert Cartwright

Robert Cartwright, Chairman
Professor of Computer Science

Walid Taha

Walid Taha
Professor of Computer Science

David W. Scott

David W. Scott
Noah Harding Professor of Statistics

Houston, Texas

April 2011

A Framework for Testing Concurrent Programs

Mathias Guenter Ricken

Abstract

This study proposes a new framework that can effectively apply unit testing to concurrent programs, which are difficult to develop and debug. Test-driven development, a practice enabling developers to detect bugs early by incorporating unit testing into the development process, has become wide-spread, but it has only been effective for programs with a single thread of control. The order of operations in different threads is essentially non-deterministic, making it more complicated to reason about program properties in concurrent programs than in single-threaded programs. Because hardware, operating systems, and compiler optimizations influence the order in which operations in different threads are executed, debugging is problematic since a problem often cannot be reproduced on other machines. Multi-core processors, which have replaced older single-core designs, have exacerbated these problems because they demand the use of concurrency if programs are to benefit from new processors.

The existing tools for unit testing programs are either flawed or too costly. JUnit, for instance, assumes that programs are single-threaded and therefore does not work for concurrent programs; ConTest and rstest predate the revised Java memory model and make incorrect assumptions about the operations that affect synchronization. Approaches such as model checking or comprehensive schedule-

based execution are too costly to be used frequently. All of these problems prevent software developers from adopting the current tools on a large scale.

The proposed framework (i) improves JUnit to recognize errors in all threads, a necessary development without which all other improvements are futile, (ii) places some restrictions on the programs to facilitate automatic testing, (iii) provides tools that reduce programmer mistakes, and (iv) re-runs the unit tests with randomized schedules to simulate the execution under different conditions and on different machines, increasing the probability that errors are detected.

The improvements and restrictions, shown not to seriously impede programmers, reliably detect problems that the original JUnit missed. The execution with randomized schedules reveals problems that rarely occur under normal conditions.

With an effective testing tool for concurrent programs, developers can test programs more reliably and decrease the number of errors in spite of the proliferation of concurrency demanded by modern processors.

Acknowledgments

I would like to thank my advisor, Robert "Corky" Cartwright. Without his guidance, advice and patience, this project would not have been possible. I thank Corky for his support and understanding, given in regard to this project as well as in private matters.

I owe many thanks to the members of my thesis committee, Walid Taha and David Scott, for their crucial suggestions, comments, and encouragements. I am grateful for their flexibility and understanding for when things really came down to the wire.

I thank Dung Nguyen, Stephen Wong and Joe Warren for getting me interested in research, and I thank Edwin Westbrook and Jun Inoue for months of tremendously enjoyable collaboration (and distraction).

I also owe a debt of gratitude to the other members of the Programming Languages Team at Rice, to the Computer Science department in general, and to my many friends. I could not have done this without you.

I thank Diana for making sure that I take care of myself. Who would have thought that I finish graduate school in the best shape I have ever been in?

Finally, I thank my mother. Words cannot express the love and respect I feel when I think of you. Everything that is good in me can be traced back to you.

It has been a long journey. Thank you for continuing to believe in me, Mama. Ich liebe Dich, so wie ein Sohn nur die beste Mutter der Welt lieben kann.

Whatevery do, or dream you can, begin it.

Boldness has genius and power and magic in it.

(Johann Wolfgang von Goethe)

Contents

Figures

Tables

Listings

Chapter 1

Introduction

In test-driven development, tests are created for a unit of code before the code itself is written, and all tests must succeed before a new revision can be submitted to the shared repository, facilitating the early detection and repair of program defects [15]. Unit tests also provide a safe foundation for refactoring the program, prevent bugs from reappearing, and can serve as documentation. The test-driven approach to software development is gaining popularity both in computer science education [15] and industrial practice [36, 5].

Unfortunately, unit testing is much less effective for programs with multiple threads of control than for sequential programs. The importance of concurrent programming, however, is rapidly growing as multi-core processors replace older single core designs, and the primary way of increasing the speed of computation is not achieved by higher CPU clock frequencies but by executing several computations in parallel. Unless there is a breakthrough in processor design or language implementation technology, writing and testing concurrent code will become a skill that all programmers must master.

Furthermore, multi-threading not only occurs when trying to utilize multi-core CPUs. Graphical user interface (GUI) frameworks like AWT, Swing, and SWT access components and react to user input in a separate event thread; therefore, GUI applications written using these frameworks already involve multi-threading.

Developers of large Java applications like DrJava [31] have identified two obstacles to applying test-driven development to concurrent programs: (i) the standard unit testing frameworks make it easy to write bad tests, and (ii) thread scheduling is non-deterministic and machine-specific, implying that the outcome of a test can change from one run to the next [35].

Test-driven design increases programmer confidence [40], which is especially important in introductory programming courses. The fact that tests with failed assertions may succeed is particularly troubling, because it could give students a false sense of security. It is therefore crucial to identify how concurrent unit tests may report false successes and what can be done to address this issue.

1.1 Motivation

This work was motivated by the problems the programmers of DrJava [31] face during the development of such a large program with many threads of control. The goal was to provide a set of tools that help developers write better concurrent software by assisting them in writing simpler, better unit tests that can be executed under different schedules. The framework should not remain a theoretical exercise but be of immediate use to programmers.

1.1.1 Thesis Statement

The previous sections highlighted some of the difficulties of developing and testing concurrent programs.

I claim that concurrent programming is difficult and not well supported by today's tools. This framework simplifies the task of developing and debugging concurrent programs.

1.2 Review of Prior Work

The following sections discuss existing (i) frameworks for unit testing, (ii) attempts to test programs under different schedules, (iii) tools for logging, and (iv) invariant checking frameworks.

1.2.1 Review of Unit Testing Frameworks

The two most widely used unit testing frameworks for Java are JUnit [18] and TestNG [50]. While TestNG provides some features that JUnit does not offer, such as dependent and data-driven tests, neither of the two frameworks includes any additional support for addressing the problems posed by concurrency.

Listing 1.1 shows a simple unit test using JUnit 3.8.2 that tests the `Square.get` method, provided in listing 1.2. The unit test shows a typical list of assertions for this kind of method: A couple of positive integers, a positive non-integral number, and a negative number are used as inputs to the function.

The expected output of each function call is given as first argument to `assertEquals`, the actual result of the function call as second argument, and since this test has to work with inexact floating-point arithmetic, the acceptable deviation from the expected value is given as third argument.

Together with the `Square` class shown in listing 1.2, the unit test will pass since all assertions are met. If, however, an incorrect implementation of squaring is used,

```
import junit.framework.TestCase;
public class SquareTest extends TestCase {
  public void testSquare() {
    // parameters are: expected value, actual value
    // and acceptable deviation (delta)
    assertEquals(4.0, Square.get(2), 0.001);
    assertEquals(9.0, Square.get(3), 0.001);
    assertEquals(6.25, Square.get(2.5), 0.001);
    assertEquals(16.0, Square.get(-4), 0.001);
  }
}
```

Listing 1.1: A Simple Unit Test

```
public class Square {
  public static double get(double x) {
    return x*x;
  }
}
```

Listing 1.2: A Simple Class to Test

like the one shown in listing 1.3, one or more assertions, and the unit test as a whole, will fail. In the particular example in listing 1.3, the function works correctly for integral input values only, and the assertion expecting a result of `6.25` for the input value `2.5` will fail.

This is only a simple demonstration of unit testing and perhaps seems unnecessary. The experience with production programming for DrJava [31], however, has shown that unit testing simple methods is an invaluable tool for ensuring that more complicated, composed methods function correctly.

JUnit 3.8.2 and older versions impose some restrictions on the developer: All methods to be run as unit tests have to be `public void` and their names have to begin with `test`. The class containing the methods also has to be a subclass of `junit.framework.Test`. TestNG and JUnit since version 4.0 freed the developer

```
public class Square {                                                    1
  public static double get(double x) {                                   2
    // note: this only works for integral values of x!                   3
    return (int)(x*x);                                                   4
  }                                                                      5
}                                                                        6
```

Listing 1.3: A Flawed Class to Test

```
import static org.junit.Assert.assertEquals;                             1
import org.junit.Test;                                                   2
public class SquareTest4 { // no need to subclass                        3
  @Test // annotation to mark test methods                               4
  public void doSquare() { // test prefix not required                   5
     assertEquals(4.0, Square.get(2), 0.001);                            6
     assertEquals(9.0, Square.get(3), 0.001);                            7
     assertEquals(6.25, Square.get(2.5), 0.001);                         8
     assertEquals(16.0, Square.get(-4), 0.001);                          9
  }                                                                      10
}                                                                        11
```

Listing 1.4: A Unit Test Using JUnit 4.2

of these limitations by using Java annotations to mark test methods. The disadvantage of this approach is lost compatibility with Java versions prior to 5.0. We have provided improved versions of JUnit 3.8.2 and 4.2, so programmers using Java 1.4 or older can still benefit from this work, while programmers using Java 5.0 or newer may claim all the advantages provided by JUnit 4.2. Listing 1.4 shows the same unit test from listing 1.1 written using JUnit 4.2.

Recently, both JUnit and TestNG gained the ability to run multiple tests, or multiple instances of the same test, in parallel. The libraries jconch [11] and parallel-junit [21] add this feature to older versions of JUnit. Running tests in parallel can shorten the testing time on multi-core machines and in some cases reveal bugs that only occur during concurrent execution. These parallel extensions, how-

```
import junit.framework.TestCase;

public class TestInOther extends TestCase {
  public void testException() {
    new Thread(new Runnable() {
      public void run() {
        // should cause failure but does not
        throw new RuntimeException();
      }
    }).start();
  }
}
```

Listing 1.5: Uncaught exception in a thread other than the main thread

ever, still ignore the fundamental flaws of JUnit and TestNG in detecting errors in multi-threaded code, such as uncaught exceptions in spawned threads.

When a Java program throws an exception, the Java Virtual Machine (JVM) unwinds the stack of the thread in which the exception was thrown until a suitable catch block is found. If no such catch block exists and the stack unwinds completely, the thread is terminated. Unit testing frameworks for Java employ a `catch (Throwable t)` block to detect uncaught exceptions in the main test thread and report failure. Since test assertions in these frameworks are implemented using exceptions, our discussion of uncaught exceptions also covers failed assertions.

This catch block only applies to the test's main thread. Since Java threads by default do not have uncaught exception handlers installed, exceptions thrown in other threads are ignored. Listing 1.5 contains a JUnit 3.8.2 test case that demonstrates that uncaught exceptions in threads other than the main thread are ignored.

Concurrency is ubiquitous in Java programs because multiple threads are required to support responsive user interfaces. Nearly all non-trivial applications with a GUI (graphical user interface) involve multi-threading. GUI frameworks

like AWT, Swing, and SWT rely on an event-handling thread to process all GUI input events and to access and update GUI components. The contracts for most AWT, Swing, and SWT methods stipulate that the method must be executed in the event-handling thread. Hence, unit tests that manipulate GUI components (e.g., creating and modifying Swing documents) must run some code in the event-handling thread. In fact, essentially all non-trivial method calls on these objects must run in the event thread. If a call on a GUI component method in the event thread is erroneous, the method may throw an exception indicating an error, but JUnit completely ignores this exception and reports success as long as no exceptions are thrown in the main thread. Similarly, a JUnit test may attach a listener to a GUI component (e.g., add a `DocumentListener` to a `Document`) to perform tests whenever the listener is fired. Even when such a listener explicitly calls the `fail` method, JUnit will not report failure because the exception generated by the call is not thrown in the main thread. (The listener is executed as a postlude to calling a method in the corresponding GUI component, which must be done in the event thread.)

Listing 1.5 exhibits another flaw, one that can also lead to a successful test, even though an uncaught exception is thrown: There is no guarantee that the child thread will reach the point of failure before the main thread has finished and the test ends. This situation is depicted in Figure 1.1.

A correctly written test ensures that all child threads have terminated before the test ends, guaranteeing that the test is aware of any uncaught exceptions thrown in child threads before the test result is determined. Java's `Thread.join` method can be used to suspend the test's main thread until a spawned child thread has finished executing. Figure 1.2 displays the behavior of a correctly written test.

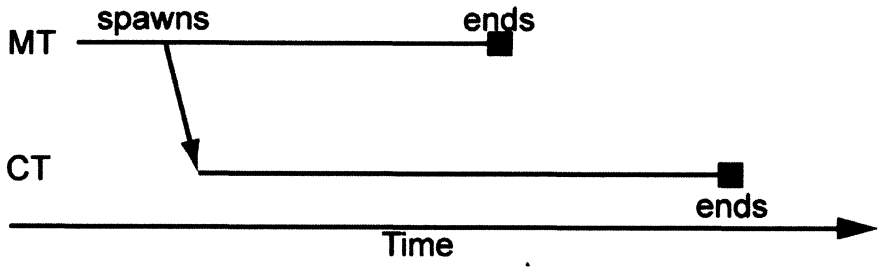

Figure 1.1 : Child thread CT outlives test's main thread MT

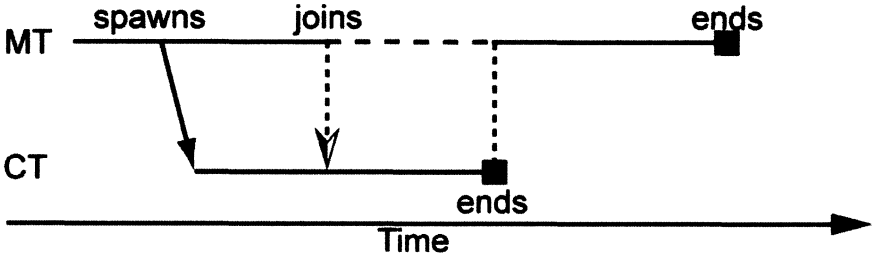

Figure 1.2 : Main thread MT joins with child thread CT

The source code for such a test can be found in Listing 1.6. Existing frameworks do not attempt to ensure that child threads terminate before the test finishes.

Even if a framework ensure that all child threads have terminated at the time the test ends, the framework still ignores the common problem that a test fortuitously succeeds even though it did nothing to enforce that the main thread finishes last. A test exhibiting this behavior is depicted in Figure 1.3. Unit testing frameworks should ensure that all threads were in fact joined and did not just terminate due to happenstance.

```
import junit.framework.TestCase;

public class MainJoinsChild extends TestCase {
  public void testException() {
    Thread child = new Thread(new Runnable() {
      public void run() {
        // exception detected with ConcJUnit
        throw new RuntimeException();
      }
    });
    child.start();
    while(child.isAlive()) {
      try {
        child.join(); // wait until child done
      }
      catch(InterruptedException ie) {
        // interrupted while waiting
        // child may not be done yet
      }
    }
  }
}
```

Listing 1.6: Main thread joins with child thread, Exception in Child Thread Detected

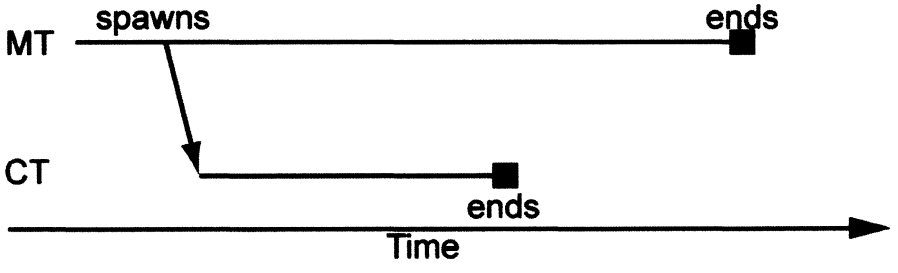

Figure 1.3 : Child thread CT ends before main thread MT, but without join

1.2.2 Review of Schedule-Aware Testing Projects

Even when the unit testing frameworks discussed in the previous section are modified to detect uncaught exceptions in all threads; detect child threads that outlive the test's main thread; and detect threads that ended on time but were not joined, the frameworks will not detect all of those problems that *could* occur; only the problems that actually occurred in the chosen schedule are found.

Even if the test suite passes unit testing without failures or warnings, it is still possible that the unit tests fails during the next run. This is due to the non-deterministic nature of thread scheduling.

Conventional unit testing assumes the program behavior is deterministic, a property that is lost for concurrent programs. The ordering of competing accesses to shared data is non-deterministic – even when those accesses are synchronized. The Java Memory Model [46] does not even guarantee sequential consistency for programs with data races, that is, the Java Memory Model does not ensure that the program execution corresponds to a serial interleaving of its threads unless the program is free of data races. The simplest strategy for avoiding data races in Java is to mark all shared variables as final or volatile. Both dynamic and static data race detectors have been developed for Java [12, 27], but this is still an active area of research.

Since concurrent program execution is non-deterministic, a unit testing framework should ideally run each test under all possible schedules. Unit testing combined with model checking can achieve this [49]: Unit tests are embedded into stub programs and then run using Java PathFinder (JPF) [54], which explores all possible schedules of the program.

The problem of executing all schedules is intractable for large programs; however, there are several mitigating factors that reduce the impact of exploring all possible schedules.

- Unit tests are written to be non-interactive, so it is not as crucial to provide results immediately as it is with programs that undergo acceptance testing. This allows unit tests to be run unattended over night. While running tests over night does not reduce the number of schedules, this possibility increases the developers' tolerance to delays that a large number of schedules creates.

- Most importantly, proponents of a model checking approach claim that unit tests are often much smaller than entire applications, resulting in a significant drop of possible schedules [49].

The second claim is false; unit tests are not "in general very short". In the experience of developers of DrJava [31], unit tests use and test program parts of any scale, not just small portions. The code in the test methods themselves may be short, but many tests exercise major subsystems, and some tests even execute the entire program, resulting in an enormous state space that needs to be explored.

By requiring a certain locking discipline that governs the way shared data is accessed, the number of schedules that need to be explored can further be reduced:

- If each shared variable is protected by at least one lock when accessed, then it is sufficient to test all interleavings of *critical blocks* between accesses to shared variables, acquiring or releasing locks, and other actions that influence concurrent behavior [6]. Adherence to this so-called mutual exclusion locking discipline can be verified by running a lockset algorithm in parallel [37].

Unfortunately, even when considering just critical blocks, the number of possible schedules increases exponentially with the size of the program and the number of threads. To simplify the calculation, let us assume that all threads contain the same number of critical blocks, and that they do not interact with each other. We also simplify the model by assuming that the scheduler does not have to be fair and is free to schedule the threads in any order. On an actual system, the scheduler is more constrained and the calculation more complex.

Then let t be the number of threads running concurrently, and b be the number of critical blocks per thread. The number of possible schedules N can then be calculated as a product of b-combinations. The formula for N, given in figure 1.4, has been derived by first choosing the b critical blocks, out of the total tb critical blocks, during which the first thread executes; there are $\binom{tb}{b}$ different choices. Then the b critical blocks for the second thread are chosen, out of the remaining $tb - b$ ones; there are $\binom{tb-b}{b}$ ways to do that. This process continues until only b critical blocks remain for the last thread. As a result, $N = \frac{(tb)!}{(b!)^t}$ [34].

Stirling's approximation, $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$, demonstrates that the number of schedules grows exponentially with both the number of threads t and the number of critical blocks b: The square root factor grows polynomially; in the second factor, both the numerator and the denominator grow exponentially, but the numerator dominates.

For example, if there are two threads, and each thread consists of two critical blocks ($t = 2, b = 2$), then there are $\frac{4!}{(2!)^2} = \frac{24}{4} = 6$ different schedules. If there are three threads ($t = 3, b = 2$), there are already $\frac{6!}{(2!)^3} = \frac{720}{8} = 90$ different schedules. It becomes apparent that the execution of all schedules quickly becomes

$$
\begin{aligned}
N &= \prod_{x=0}^{t-1} \binom{(t-x)b}{b} \\
&= \binom{tb}{b}\binom{(t-1)b}{b}\binom{(t-2)b}{b}\cdots\binom{2b}{b}\binom{b}{b} \\
&= \frac{(tb)!}{b!(tb-b)!}\frac{((t-1)b)!}{b!(tb-2b)!}\frac{((t-2)b)!}{b!(tb-3b)!}\cdots\frac{(2b)!}{b!b!}\frac{b!}{b!\,0!}
\end{aligned}
$$

The second term in the denominator of one fraction cancels out the term in the numerator of the next fraction, resulting in the simpler term.

$$
\begin{aligned}
&= \frac{(tb)!}{b!}\frac{1}{b!}\frac{1}{b!}\cdots\frac{1}{b!}\frac{1}{b!} \\
&= \frac{(tb)!}{(b!)^t}
\end{aligned}
$$

Figure 1.4 : Formula for Number of Schedules

intractable, even if context switches are only modeled where operations can affect other threads.

A practical alternative to exhaustively running each test under all schedules might be to run each test under a set of randomized schedules. There are several projects that employ randomized testing: ConTest [9] by IBM's Verification and Testing Technologies (STAR) group in Haifa, Israel; rstest [42] by Scott Stoller at Stony Brook University; and CalFuzzer [38, 17] by Koushik Sen *et al.* at University of California, Berkeley.

All three products perform bytecode rewriting to insert additional instructions at all synchronization points, that is, in all the places that could affect scheduling. Typically, these instructions will be calls to `Thread.sleep`, `Thread.yield`, or `Thread.setPriority`. ConTest also includes a record-and-replay facility that allows the schedule that was actually executed in a certain run to be recorded and replayed again.

rstest is similar to ConTest, but uses fewer insertion sites, resulting in faster executions. Furthermore, rstest claims to exhibit *probabilistic completeness*, meaning there is a non-zero probability of finding every assertion violation.

CalFuzzer is a framework for developing analyses for concurrent programs and provides callbacks when synchronization operations or memory accesses are executed. After using an imprecise analysis of the program to discover potential problems, CalFuzzer perturbs the scheduling to reveal actual concurrency bugs.

Unfortunately, ConTest and rstest predate the Java Memory Model [46], and the literature on CalFuzzer does not mention the Java Memory Model. The three products therefore make incorrect assumptions about the operations that affect synchronization: They assume that programs exhibit an "as-if-serial" execution, but the Java Memory Model only provides such a guarantee of sequential consistency in the absence of data races, as defined according to the *happens-before* relation of the Java Memory Model). Many concurrency defects *introduce* data races to the program; therefore, assuming the sequential consistency of an "as-if-serial" execution is not appropriate in practice.

Assuming sequential consistency also calls into question the usefulness of ConTest's record-and-replay mechanism and rstest's probabilistic completeness property. It is not clear how accurately a schedule can be recorded and reproduced if the program executes on multiple processors as opposed to being time-sliced on a single processor. The lack of an "as-if-serial" schedule to which the actual execution with data races can be mapped may make it impossible to replay the schedule as it happened. Furthermore, there is a general problem with adding code to a program to record and replay its execution: The additional instructions may modify the sched-

ule and cause the program to execute under a different schedule than it normally would.

The probabilistic completeness property provided by rstest also relies on a sequentially consistent execution. At every synchronization point, rstest assumes that there is "a non-zero probability of transferring control to each runnable thread." [42] Such a possibility of transferring control would allow rstest to execute even schedules that the particular scheduler normally does not allow. In a program with data races, for which the Java Memory Model does not guarantee sequential consistency, there is no total order over all operations, and several operations may appear to execute at the same time. In those cases, it is not possible to control the order of operations, and probabilistic completeness is lost.

The availability of ConTest and rstest is also quite limited. ConTest is not freely available from IBM, and ConTest for Java is only temporarily available as a trial version while it is being alpha-tested. rstest is not freely available at all.

Furthermore, none of the examples used in the literature on ConTest or rstest are available as source code. ConTest only supplied four simple experiments [9], but they were briefly described in prose*. rstest provides two similarly small examples, also not available as source and only described in prose [42]. In addition to those two examples, rstest uses Stephen Hartley's Java version of the xtango Animation Library [41] as third example; unfortunately, this version is not available anymore. The fourth example rstest uses is an unspecified version of the ArgoUML application [3], which was tested for several minutes with "with semi-random manual inputs" [42].

---

*`FundManager`, an apparent semantic match of ConTest's second example, was eventually retrieved [48].

The limited availability of ConTest and rstest, as well as the lack of sample programs, makes it nearly impossible to accurately compare this work, the ConcJUnit framework, to them. The effectiveness of Concutest will therefore be determined by itself.

CalFuzzer is the most recent of the related works and is available from the Berkeley website [7]. It focuses on different concurrency problems at a time, though, such as atomicity violations, data races, or deadlocks.

CalFuzzer appears most promising, provided it conforms to the Java Memory Model. It does not provide the kind of comprehensive testing framework that is part of this work. Concutest includes crucial improvements to JUnit, as well as a logging framework and an invariant checker; CalFuzzer does not.

1.2.3 Review of Execution Logging Frameworks

Unit tests often need to determine whether certain portions of code have executed. In traditional functional programs, this can simply be done by checking the return value. Listing 1.7 shows an example of such a method with a return value, and listing 1.8 shows the unit test that invokes the method, checks the return value, and thereby also verifies that the method has been executed.

```
public class LoggingResult {                                    1
  public int computeSomething() {                               2
    return 123;                                                 3
  }                                                             4
}                                                               5
```

Listing 1.7: Application code with a method that has a return value

Unfortunately, checking whether a method has been executed becomes more complicated when there is no return value. This is often the case in event-driven systems such as the GUI frameworks AWT, Swing, and SWT, where application code is invoked as a result of a change in the GUI. More generally, the problem of logging the execution of code without return value frequently occurs when the command, strategy, and observer design patterns [13] are used. Swing's `Runnable` interface is an example of the command design pattern; the strategy design patterns can be found in Swing in the form of the many pluggable look and feel (PLAF) UI classes, such as `TreeUI`, that govern the behavior of GUI components; and the observer design pattern is embodied by the many listener interfaces, such as `Action Listener`.

Generally, this problem is solved by adding a flag to the application code that is set to true when the method in question has executed. That flag is then checked in the test code. Listing 1.9 shows a method without a return value that sets a flag to `true`; the test code listing 1.10 then ascertains that the flag is indeed `true`, implying that the method has been executed.

```
import junit.framework.TestCase;                                    1
                                                                    2
public class LoggingResultTest extends TestCase {                  3
  public void testResult() {                                        4
    LoggingResult app = new LoggingResult();                        5
    int result = app.computeSomething();                            6
    assertEquals(123, result);                                      7
  }                                                                 8
}                                                                   9
```

Listing 1.8: Test code for a method that has a return value

```
public class LoggingFlag {
  public volatile boolean hasExecuted = false;

  public void noResult() { // no return value
    hasExecuted = true;
    // do something
  }
}
```

Listing 1.9: Application code with a method that does not have a return value, but that uses a flag

```
import junit.framework.TestCase;

public class LoggingFlagTest extends TestCase {
  public void testFlag() {
    LoggingFlag app = new LoggingFlag();
    app.noResult(); // no return value
    assertTrue(app.hasExecuted);
  }
}
```

Listing 1.10: Test code for a method that does not have a return value, but that uses a flag

This solution is not desirable, as it introduces artifacts only necessary for the execution of the tests into the application code. As a result, the application and its test become tightly coupled.

More general logging frameworks, such as Oracle's Java Logging Technology (`java.util.logging`) [30] or Apache's log4j [1], intended to produce human-readable output, could also be used to test whether a certain piece of code has been executed, but they suffer from the same problem of introducing tight coupling and, due to their verbose text-based nature, would be less efficient.

To keep application code and test code decoupled, there should be no test artifacts in the application. This can be achieved using aspect-oriented programming [55]. In AspectJ [51], for example, the execution of a method without return value could be logged by writing an *aspect* that defines a *pointcut* specifying the method that should be logged. The aspect then defines an *advice* to be inserted, or *woven* in, before the method's body executes. In this advice, a flag is set to `true`, and that flag is later checked in the test. Listing 1.11 shows the application code without any flags. Listing 1.12 shows the aspect that inserts the advice at the beginning of the method. Listing 1.13 shows the test containing the flag.

In this AspectJ example, all test-related artifacts are found either in the aspect or the test code, but not in the application code. This solution is elegant and general, but it has some disadvantages as well. First, it requires the use of a different language and a different compiler, namely AspectJ. This alone may be enough to dissuade developers from using aspect-oriented programming for logging.

Furthermore, AspectJ is limited in what classes it can directly modify. Furthermore, to allow advice to be woven into library classes of the Java Development Kit (JDK), AspectJ duplicates code [4, 52, 53]. This is necessary to avoid infinite recur-

```
public class LoggingAJ {
  // no flag
  public void noResult() { // no return value
    // do something
  }
}
```

Listing 1.11: Application code with a method that does not have a return value and does not use a flag (using AspectJ)

```
public aspect LoggingAJAspect {
  pointcut methodToLog() :
    execution(public LoggingAJ.noResult());

  before() : methodToLog() {
    // set flag
    LoggingAJTest.hasExecuted = true;
  }
}
```

Listing 1.12: Aspect to insert logging code into a method that does not have a return value and that does not use a flag (using AspectJ)

```
import junit.framework.TestCase;

public class LoggingAJTest extends TestCase {
  // flag in test code
  public static synchronized boolean hasExecuted = false;
  public void testAspectJ() {
    LoggingAJ app = new LoggingAJ();
    app.noResult(); // no return value
    assertTrue(hasExecuted);
  }
}
```

Listing 1.13: Test code for a method that does not have a return value and that does not use a flag (using AspectJ)

sion, since the advice may use the very same library classes as well. The duplication yields unparalleled generality for weaving any advice into any class, but also brings code bloat with it that is unnecessary for the use as a logging framework.

The logging framework developed as part of Concutest, which uses Java annotations and minimal bytecode rewriting, avoids these problems: It uses the Java language and the Java compiler, and it is not necessary to keep duplicates of code.

A proposed extension of the Java language allowing subtyping for annotations, discussed in section A and implemented in the xajavac compiler [32], allows for even cleaner specification of the methods whose execution should be logged.

1.2.4 Review of Invariant Checking Frameworks

The proposed Java language extensions for annotation subtyping, discussed in section A.2, can also be leveraged in the invariant checking framework provided by Concutest.

Though Concutest can check arbitrary invariants in the form of preconditions, its annotations are primarily concerned with concurrent programming and maintaining a *threading discipline.* Just as important as testing concurrent programs is the task of defining, documenting, and enforcing a set of rules, or threading discipline, that dictate which threads must acquire what sets of locks before they may access data.

Examples of threading discipline can easily be found, for example in the Swing GUI libraries:

- The Javadoc documentation for Java's `TreeModel`, `DefaultTreeModel`, `TreeNode`, `MutableTreeNode`, and other classes related to Swing's tree model states that all the methods first defined in them may only be called from the

event thread. The same applies to classes that belong to Swing's table model, and some methods involved in model-to-view coordinate conversion.

- In contrast, Java's `SwingUtilities.invokeAndWait` method may not be called from the event thread, since doing so would lead to an instantaneous deadlock.

- Before making a destructive or compound operation on instances of Java's `AbstractDocument` class a read or write lock has to be acquired and, after the operation, be released. A read lock acquired and released using the `readLock` and `readUnlock` methods, while a write lock uses the `writeLock` and `writeUnlock` methods, respectively.

These invariants define threading discipline, and they all came just from the Swing GUI library; applications and other libraries usually have their own disciplines that need to be followed.

More generally, an invariant can be any kind of proposition that must be true when a method is entered (a precondition) or exited (a postcondition). Concutest only supports preconditions, but other invariant checking frameworks support both.

In its simplest form, invariants can be checked using assertions. Listing 1.14 uses Java's `assert` statement to check pre- and postconditions.

One of the benefits of using an `assert` statement is that it can be disabled in release builds; therefore, it does not have any impact of the runtime of the application. This absence of a detriment to program performance avoids the problem of undesirable entanglement of application code and test code that was found to exist with many logging solutions discussed in the previous section.

There still are test artifacts in the application code, namely the stated pre- and postconditions, but explicitly stating them is often even viewed as beneficial to program quality. Doing so forms the core of the *Design by Contract* software engineering practice [26], the roots of which go back to Hoare's axiomatic basis for computer programming [14].

One problematic aspect of using Java's assert statement to check invariants is that the invariants are not inherited by methods that are overridden in subclasses. Listing 1.15 shows a class `C` whose method m has the precondition that the argument `a` is not zero. When the method `m` is overridden in class `D`, which is a subclass of class `C`, the precondition is not present anymore.

Existing invariant checking frameworks such as jContractor [20, 19], CoJava [22], or the contract compiler by Findler *et al.* [10] allow the programmer to express rich propositions as pre- and postconditions, and often postconditions may even reference the old state of an object at the time of method entry.

To a certain degree, the existing invariant checking frameworks also correct the problem of inheriting invariants: Unless the invariants are modified for an overridden method, the overridden method has to adhere to the same contract as the original method. This is useful in object-oriented programs that make frequent use of subclassing. Compared to using `assert` statements, automatically inheriting invariants from methods in superclasses is a definite benefit.

However, all invariant checking frameworks that were studied nonetheless suffer from at least one defect: They do not rely on the Java language alone, but rather extend the language with additional constructs or place the preconditions in comments, which are not checked by the Java compiler.

```
public class InvariantAssert {
  private int divisor = 0;
  public int divideAndSet(int dividend) {
    assert (divisor != 0); // precondition
    try {
      divisor = dividend / divisor;
      return divisor;
    }
    finally {
      assert (divisor != 0); // postcondition
    }
  }
}
```

Listing 1.14: Using assert to check pre- and postconditions

```
class C {
  public void m(int a) {
    assert (a != 0); // precondition
    // do something
  }
}

class D extends C {
  public void m(int a) {
    // no precondition here
    // do something else
  }
}
```

Listing 1.15: assert statements are not inherited by methods overridden in subclasses

With the exception of Findler's work [10], the frameworks also improperly synthesize the invariants for overridden methods if the contracts are changed.[†]

For instance, all frameworks except Findler's synthesize the precondition for an overridden method using a disjunction. For this example, consider the `m` methods in listing 1.16. In the base class `C`, the programmer declares $x > 0$ as precondition. In the overridden method, the precondition $x > 10$ is added. Most frameworks synthesize $(x > 0) \vee (x > 10)$ as precondition for the overridden method.

This synthesized precondition is problematic since it may violate the behavioral subtyping condition [25]. In object-oriented systems, it should be possible to use an overridden method in a subclass anywhere the method in the superclass was used. More formally, that means the precondition may be made harder to fulfill, and the postcondition may be weakened. Let $p^C(x)$ and $p^D(x)$ be the preconditions on the methods `C.m` and `D.m`, respectively, where `D extends C`. Similarly, let $q^C(x)$ and

[†]The literature about jContractor [19] points out the problem of synthesizing correct invariants, but jContractor does not correctly implement invariants for overridden methods either, as pointed out by Findler [10].

```
class C {                                                    1
  @Pre("x > 0")                                              2
  void m(int x) { ... }                                      3
}                                                            4
                                                             5
class D extends C {                                          6
  // this precondition should be rejected                    7
  // "x > 0" does not imply "x > 10"                         8
  @Pre("x > 10")                                             9
  void m(int x) { ... }                                      10
}                                                            11
```

Listing 1.16: Bad additional precondition in overridden method.

$q^D(x)$ be the respective postconditions on the methods `C.m` and `D.m`. Then behavioral subtyping requires:

$$\forall x, p^C(x) \Rightarrow p^D(x)$$
$$\forall x, q^D(x) \Rightarrow q^C(x)$$

If a disjunction is used for the synthesized precondition of method `D.m`, then the implication is trivially fulfilled, even though the overridden method `D.m` cannot be used everywhere the superclass method `C.m` can be used:

$$\forall x, (x > 0) \Rightarrow [(x > 0) \vee (x > 10)]$$

even though $(x > 0) \not\Rightarrow (x > 10)$. Or generally,

$$\forall x, p^C(x) \Rightarrow [p^C(x) \vee p^D(x)]$$

This kind of invalid precondition should be rejected, and Findler's work does so by (1) checking the subclass precondition alone, (2) then recursively checking the superclass precondition, and (3) finally verifying that the superclass precondition implies the subclass precondition[‡]

The invariant checker that is part of Concutest does not implement postconditions, but it does implement synthesized preconditions correctly.

1.3 Organization

The following chapters will discuss the contributions of this work.

[‡]This is done for the values of the parameters of that particular method invocation. The tool does not check that $p^C(x) \Rightarrow p^D(x)$ for all possible values of x.

Chapter 2 presents the details of ConcJUnit, the improved JUnit framework. Adding uncaught exception handlers to all threads, ensuring that child threads have terminated, and checking that child threads are properly joined are fundamental for effective testing of concurrent programs.

Chapter 3 describes a tool that adds delays and yields in critical places of the program, helping to explore different schedules. As mentioned before, the erratic nature of thread scheduling often causes concurrent programs to be nondeterministic. This means that even with the improvements from chapter 2, a test suite may pass during one run and fail the next time. Executing the tests under different schedules increases the probability that even rare errors are found.

Writing good unit tests for concurrent programs is nonetheless difficult, even when the unit testing framework executes programs using varying schedules. Concutest therefore provides additional support in the form of logging and invariant checking.

Chapter 4 describes how Java annotations can be used to specify which methods should be logged. Placing this metadata in annotations helps separate test code from application code.

Chapter 5 presents another use of Java annotations: Invariants can be encoded as annotations on methods or classes and then checked at runtime. Allowing subtyping for Java annotations, described in the appendix in section A.2, lets developers create a powerful language of invariants.

Chapter 6 explores more general aspects of bytecode rewriting, a technique that was used in all parts of this framework.

The conclusion in chapter 7 summarizes the contributions of this work and provides an outlook on future work.

Chapter 2

Improvements to JUnit

Despite the proven effectiveness of current unit testing frameworks for programs with a single thread of control, developers have found them difficult to use in multithreaded programs. Writing good unit tests for concurrent programs is hard, for several reasons:

- Thread scheduling is non-deterministic and machine-specific, so the outcome of a unit test may change from one run to the next.
- The non-determinism makes it hard to reproduce problems that only occur in particular schedules, and even harder to ensure that the unit tests pass under all possible schedules.
- Attempting to simulate a test under different schedules by adding additional locks or `wait-notify` communication between the test and the framework results in unwieldy code.

As a result, a successful unit test provides only little assurance, and only a unit test failure imparts tangible information to the developer. A failure proves that a problem exists in the program, but a unit test success does not prove that the unit test will always succeed. To make matters worse, the existing unit testing frameworks do not correctly deal with concurrent programs and may completely ignore errors that occur in threads other than the main thread. This chapter describes

three defects in current unit testing frameworks and then introduces ConcJUnit, an improved version of JUnit that does not share these deficiencies.

2.1 Default Exception Handler

The most serious problem of using JUnit to test concurrent programs is the lack of reporting uncaught exceptions in auxiliary threads. A new child thread does not have a default exception handler installed; therefore, unless a thrown exception is caught somewhere in the program, the exception will unwind the thread's stack, invoke Java's own exception handler to print a message, and then terminate the thread. Other threads are not automatically notified of this, so an uncaught exception in an auxiliary thread will be completely unnoticed by the main thread and JUnit, even though the same code executed in the main thread would lead to a test failure. Because JUnit's assertions, like `assertEquals` shown in listings 1.1 and 1.4, are implemented using exceptions, failed assertions will not be noticed either.

Listings 2.1 and 2.2 illustrate this problem in greater detail: The unit tests in both listings contain two methods, `testException` and `testAssertion`. In both listings, `testException` throws an exception that is not caught anywhere, while `testAssertion` makes an assertion that is guaranteed to fail. Therefore, in both listings 2.1 and 2.2, the two test methods should produce failures. In listing 2.2, however, an auxiliary thread throws the exception and makes the assertion, not the main thread as in listing 2.1. JUnit is never informed of the uncaught exception or the failed assertion and declares both test methods successful.

To remedy this problem, the modified ConcJUnit framework creates a new thread group with an overridden `ThreadGroup.uncaughtException` method. The

```
import junit.framework.TestCase;
public class TestInMainThread extends TestCase {
  public void testException() {
   // uncaught, causes failure
   throw new RuntimeException();
  }
  public void testAssertion() {
    // fails, causes failure
    assertTrue(false);
  }
}
```

Listing 2.1: Uncaught Exception and Assertion

```
import junit.framework.TestCase;

public class TestInOther extends TestCase {
  public void testException() {
    new Thread(new Runnable() {
      public void run() {
        // should cause failure but does not
        throw new RuntimeException();
      }
    }).start();
  }
}
```

Listing 2.2: Uncaught Exception in an Auxiliary Thread

framework then spawns a new thread in this thread group, executes the test in the new thread, and waits for the test to complete. If an uncaught exception is thrown in the test's thread or any of its child threads, the overridden `uncaughtException` method stores the exception to make it accessible to ConcJUnit. When the ConcJUnit framework resumes execution, it checks whether an uncaught exception has occurred and deals with the exception appropriately, just as if it had occurred in the main thread.

The use of a thread group is essential for two reasons:

- When a parent thread spawns a child thread, the parent's thread group is inherited by the child thread (unless a specific thread group is passed to the child's constructor; more about this below). This assigns the same thread group to auxiliary threads as to the test's main thread; therefore, uncaught exceptions in auxiliary threads also invoke the overridden `uncaughtException` method, and uncaught exceptions and failed assertions are no longer ignored.

- Before Java 5.0 introduced the `setDefaultUncaughtExceptionHandler` method, using a thread group was the only way to catch exceptions in threads other than the current thread. While the feature introduced with Java 5.0 is easier to use, thread groups offer compatibility with older versions of Java. Thread groups are also more robust than the new `setDefaultUncaughtExceptionHandler` method: There is only one default uncaught exception handler, and in order to function correctly, the ConcJUnit framework would have to prevent the program from changing it. A program that creates thread

groups, on the other hand, is not affected because of the hierarchical nature of thread groups.

Thread groups were introduced to the Java API to process threads and their descendants collectively. The ConcJUnit framework uses this feature to record uncaught exceptions in all threads that a test spawns.

There are a few problems with this approach:

1. Programmers can supply their own thread groups when creating threads, thereby overriding the thread group installed by ConcJUnit. This is normally unproblematic, since a thread group created in one of the test's threads is a descendant of ConcJUnit's thread group, which is still informed about all the uncaught exceptions it should know about. Which exceptions need to be recorded by ConcJUnit depends on where they occur and whether the `uncaughtException` method of the new thread group has been overridden:

   a. If the `uncaughtException` method has been overridden by the programmer, then the intent has been declared that the program should handle uncaught exceptions itself. ConcJUnit may therefore not record these uncaught exceptions.

   b. The only place where uncaught exceptions should be reported to ConcJUnit is in the overridden `uncaughtException` method itself. Unfortunately, uncaught exceptions thrown there do not get processed by Java at all. I believe that this is an oversight in the Java Language Specification [43] and that the parent thread group's `uncaughtException` method should be invoked.

   c. If the `uncaughtException` method has not been overridden, then the basic behavior of the `ThreadGroup.uncaughtException` method will automatically

call the `uncaughtException` method in the parent thread group, and the exceptions are correctly registered.

2. It is possible to create a new thread group that is not a descendant of the current thread group. If the programmer deliberately creates a thread group that does not descend from ConcJUnit's thread group, then exceptions could go unnoticed by the ConcJUnit framework.

3. Uncaught exceptions thrown in the `uncaughtException` method of an application's `Thread.UncaughtExceptionHandler` cannot be processed, since the Java virtual machine ignores them [45]. Again, I believe this is an oversight in the Java Language Specification [43] and that the `uncaughtException` method of the thread group should be invoked.

While the problems described in 1.b., 2. and 3. are real, the probability of accidentally ignoring uncaught exceptions is low: Most code does not use thread groups at all (in March 2007, Koders [23], a source code search engine, found 913 matches for "ThreadGroup" in the Java source code it had scanned, compared to 49,329 matches for "Thread"), does not override the `uncaughtException` method (in March 2007, Koders reported 32 method definitions as matches for "uncaughtException"), and does not create thread groups that do not descend from the current thread group. For nearly all programs, the ConcJUnit framework can report all uncaught exceptions. Furthermore, the improved framework reports all uncaught exceptions reported by the original framework.

It is important to understand that these improvements will not detect all uncaught exceptions that *could* occur; only the uncaught exceptions thrown in the chosen schedule are found. It is possible that the program can be executed under

```
import junit.framework.TestCase;                                    1
public class TestInOtherThreadSleep extends TestCase {              2
  public void testException() {                                     3
    new Thread(new Runnable() {                                     4
      public void run() {                                           5
        // sleep for 10 seconds                                     6
        try { Thread.sleep(10*1000); }                              7
        catch(InterruptedException ioe) { /* ignore */ }            8
        // uncaught, should cause failure but does not              9
        throw new RuntimeException();                               10
      }                                                             11
    }).start();                                                     12
    // test's main thread exits immediately                        13
  }                                                                 14
}                                                                   15
```

Listing 2.3: Uncaught Exception in an Auxiliary Thread Reached Too Late

a different schedule and fail. Adding a default exception handler to ConcJUnit is nonetheless a crucial step in creating a framework suitable for testing concurrent programs.

2.2 No Living Child Threads Check

If JUnit is modified as described in section 2.1, the ConcJUnit framework is now able to detect uncaught exceptions in auxiliary threads. Unfortunately, listing 2.2 exhibits another problem often found in tests of concurrent software: The test does not ensure that the auxiliary threads spawned in `testException` and `testAssertion` finish before the test ends and is declared a success. The two test methods in listing 2.2 may or may not be successful, but the test method in listing 2.3 is almost guaranteed to succeed even though it should fail: Because of the call to `Thread.sleep`, the auxiliary thread is unlikely to reach its point of failure in time.

It is clear that this problem is caused by the test's main thread not waiting for auxiliary threads to finish before the test ends. A correctly written test should ensure this by having a `Thread.join` call for every thread that it spawned; therefore, it is logical to declare as invalid a test that finishes before all of its child threads have terminated.

To address this issue and increase the framework's ability to detect problematic tests, the ConcJUnit framework checks if any child threads are still running. The framework enumerates the remaining threads in the ConcJUnit thread group and declares the test a failure if running threads are found after the test ended. To help the programmer fix the incorrectly written test case, still running child threads are listed, along with their current stack traces.

Some threads are excluded from the list of threads and are allowed to outlive the test's main thread:

- Daemon threads are automatically shut down once all non-daemon threads of an application have terminated. In a unit test, they continue to run, though, because the unit test is not a stand-alone application. It is therefore reasonable to allow daemon threads to remain active even after the test has ended.

- Some system threads, namely those part of the AWT and RMI (remote method invocation) libraries, may be created inside the thread group, but this happens automatically and without the programmer's knowledge. Just like daemon threads, they would terminate automatically once the application finishes. For the reasons explained above, these threads are allowed to remain active after the test has ended:

  - `AWT-Shutdown`

```
import junit.framework.TestCase;
public class TestInOtherThreadSleepJoin extends TestCase {
  public void testException() {
    final Thread t = new Thread(new Runnable() {
      public void run() {
        // sleep for 10 seconds
        try { Thread.sleep(10*1000); }
        catch(InterruptedException ie) { /* ignore */ }
        // uncaught, causes test to fail
        throw new RuntimeException();
      }
    });
    t.start();
    // main thread waits for spawned thread to finish
    try { t.join(); }
    catch(InterruptedException ie) { /* ignore */ }
  }
}
```

Listing 2.4: Main Thread Waits for Auxiliary Thread to Finish

- AWT event threads, i.e. threads starting with `AWT-EventQueue-`
- `RMI Reaper`
- `DestroyJavaVM`

The improvements added in this section find the active child thread in listing 2.3 and declare the test a failure. Listing 2.4 shows a correctly written version of the test: Even though the auxiliary thread sleeps for 10 seconds, the uncaught exception after the sleep is detected and causes the test to fail since the main thread waits for the auxiliary thread to terminate.

2.3 Enforcement of Thread Joins

The check for living threads described in the previous section only emits warnings for a faulty test whose main tread terminates before all child threads have finished.

For experienced developers, this case should be rare. It would therefore be useful to also detect badly written tests that happened to succeed even though they did not enforce that the main thread finishes last. A test that relies on such a fortuitous circumstance is depicted in figure 1.3.

A fork/join design in which each parent thread has to join with all of its child threads can solve this problem. Figure 2.1 demonstrates this scheme: The main threat `MT` spawns a child thread `CT1`; `CT1` itself spawns another child thread `CT2`. `CT1` cannot terminate before `CT2` has terminated, and `MT` cannot end before `CT1` has ended. Therefore, `MT` cannot end before all of its ancestor threads have finished executing.

This simple model is common in the parallel algorithms literature, but it may be too restrictive for general-purpose Java programs. For example, it should be permissible for the main thread to join directly with all of its ancestor threads, whether it started them itself or not. This is illustrated in figure 2.2. Another possibility that also ensures that all ancestor threads terminate before the test's main thread ends is to spawn a chain of helper threads, each guaranteed to outlive the previous thread. The main thread then merely has to join with the last of the helper threads. This situation is shown in figure 2.3.

The concept of a chain can be generalized into a directed acyclic graph called "join graph", initially just consisting of a node for the main thread. Every time a new child thread is spawned, a new node is added to the graph, and every time a thread `A` joins with another thread `B`, an edge from `A` to `B` is added. Such an edge indicates that `B` is guaranteed to have terminated before `A`. Therefore, to ascertain that all child threads have terminated before a test's main thread ends, the framework just need to verify that all nodes are reachable from the main thread's node

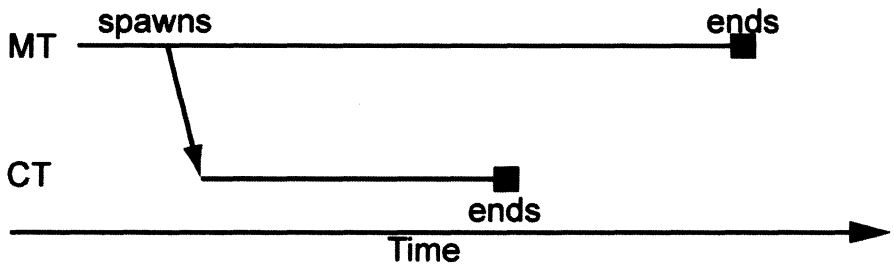

Figure 2.1 : Each parent thread joins with its child thread (`MT` joins with `CT1`, `CT1` with `CT2`)

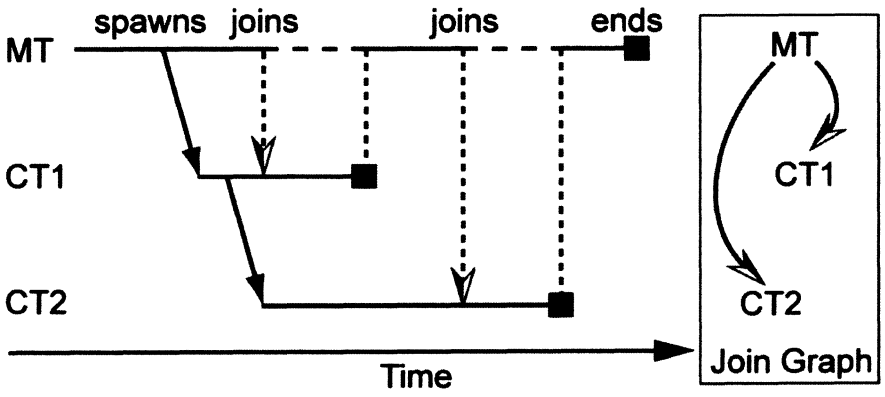

Figure 2.2 : Main thread joins with both child threads (`MT` joins with `CT1`, `MT` with `CT2`)

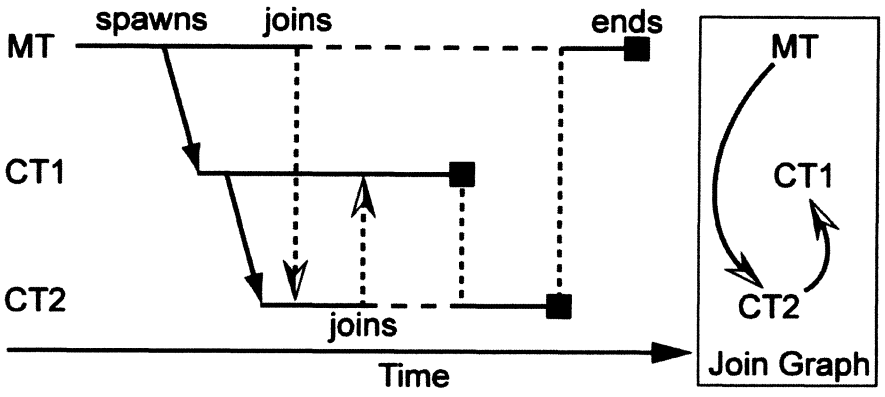

Figure 2.3 : Main thread joins with last thread in chain (`MT` joins with `CT2`, `CT2` with `CT1`)

in the join graph. Note that in figure 2.1, figure 2.2, and figure 2.3 all nodes are reachable from the main thread's node `MT`. In figure 2.4, however, where no thread joins with `CT2`, the node for `CT2` is not reachable from `MT`, indicating that a "lucky" warning should be issued.

Similar to the way ConcJUnit constructs the join graph, the framework also creates a "start graph" that records the child threads spawned by each thread: Every time a new thread is started, an edge from the parent thread to the child thread is added. This graph allows us to determine exactly which threads are ancestors of the test's main thread so ConcJUnit can ignore other threads that may have been running but were not created by the test.

While the improvements described in the previous two sections only required changes to the JUnit framework, detecting child threads that were not targets of a join operation requires modifying the Java Runtime Environment (JRE). The byte-

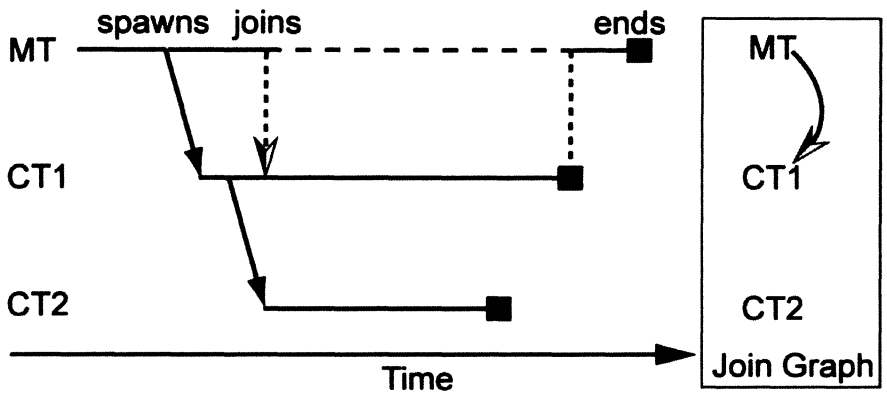

Figure 2.4 : CT2 not reachable in join graph (MT joins with CT1, CT2 not joined by any thread)

code of the java.lang.Thread class needs to be changed to perform the necessary bookkeeping at the end of the `Thread.start` and texttttThread.join methods.

Concutest contains a tool that processes the `rt.jar` file (or `classes.jar` file on Mac OS X) of the JRE the user has installed, generating a replacement `rt.jar` file containing the modified `java.lang.Thread` class and its helpers. During testing, this replacement `rt.jar` is put on Java's boot classpath using the `-Xbootclasspath/p:rt.jar` command line option. The bookkeeping is performed in a helper class `ThreadGraphs` with two static methods, `addThreadStarted(Thread t)` and `addThreadJoined(Thread t)`. Both methods have the current thread, as returned by the `Thread.currentThread` method, as implicit argument. For each thread, `ThreadGraphs` maintains a set of threads started by it and a set of threads joined by it. Together, these sets form the start and join graphs. I chose to implement these graphs as adjacency lists using `NonBlockingHashMap` and `NonBlockingHashSet` from the Highly Scalable Java [8]

library. These data structures are lock-free and therefore minimize the probability that the additional bookkeeping done by ConcJUnit changes the thread scheduling of the test. With synchronized data structures employing locks, schedule perturbations would be likely.

At the end of a test, ConcJUnit attempts to retrieve the contents of these graphs using reflection. The library is not hard-linked against the modified `java.lang.Thread` class and therefore also works without it on the boot classpath; in that case, ConcJUnit just does not emit "lucky" warnings.

2.4 Results

To test the effectiveness and performance of ConcJUnit, I replaced JUnit with ConcJUnit and executed the unit test suites for DrJava [31] (revision 4918) and JFreeChart [16] (1.0.13), an open-source library to display data visually. The extensive JFreeChart tests were not concurrent, but they all passed, underscoring the compatibility of ConcJUnit with existing code.

Of the 900 unit tests contained in the DrJava test suite, 880 tests passed without any warnings. A single test emitted a "no join" warning, and 18 tests issued "lucky" warnings regarding their join behaviors. There were no tests that failed as a result of replacing the unit testing libraries, but one test timed out.

Upon examination of the source code, the 18 "lucky" warnings and the single "no join" warning were all legitimate. The "no join" warning was issued during a test that created a remote process that did not terminate during the test, causing the thread waiting for the termination to outlive the test.

The "lucky" warnings were emitted by tests that in fact did not join with all the child threads they had spawned. Instead, they used a wait-notify scheme to ensure that the child threads terminate before the tests end. In all of these cases, the developers had taken care that there were no more lengthy operations after the notifications, and that an uncaught exception after the call to notify was unlikely. This practically makes the wait-notify scheme equivalent to a join. However, if additional work were to be performed after the notification, and if one of the operations were to fail, such a test could be incorrectly declared a success.

During my examination, I did not discover any tests that ignored uncaught exceptions or failed assertions in spawned threads, but for DrJava, a mature project built with test-driven methods, this was expected. Using ConcJUnit allowed us to replace the handler for uncaught exceptions that was custom-built for DrJava with the general one found in ConcJUnit. Doing this also made a test of the exception handler redundant, eliminating one of the "lucky" warnings.

The overhead, introduced by ConcJUnit to handle uncaught exceptions in all threads and to detect the "no join" and "lucky" conditions, was barely perceptible. The total slowdown for ten runs of the entire test suite was 55.2 seconds, or 1.1 percent of the 5252.4 seconds it took to run the entire suite ten times using JUnit.

It was easy to integrate ConcJUnit into the existing DrJava project and use it instead of the original `junit.jar`. I did not expect many problems, since the DrJava developers had already created a much more complicated solution to deal with multithreading and exceptions in other threads.

ConcJUnit is also available to users of DrJava. Beginning with the drjava-beta-20100507-r5246 release, users can select four levels of concurrency support for unit tests: (1) old JUnit, (2) ConcJUnit with exception handlers, (3) ConcJUnit with ex-

ception handlers and "no join" warnings, or (4) full ConcJUnit with exception handlers, "no join" warnings, and "lucky" warnings.

If the DrJava user wants to generate "lucky" warnings, then DrJava automatically generates the modified `rt.jar` or `classes.jar` file.

It has always been possible to write concurrent unit tests, but it has been very difficult to write them well. Despite the complicated DrJava code and extensive unit tests, the ConcJUnit library was able to detect several flaws.

```
Thread Timer-0 (java.util.TimerThread) is still alive: state=
   WAITING
        java.lang.Object.wait(Native Method)
        java.lang.Object.wait(Object.java:474)
        java.util.TimerThread.mainLoop(Timer.java:483)
        java.util.TimerThread.run(Timer.java:462)
Thread Wait for Interactions to Exit Thread (edu.rice.cs.util.
   newjvm.AbstractMasterJVM$1) is still alive: state=RUNNABLE
        java.lang.ProcessImpl.waitFor(Native Method)
        edu.rice.cs.util.newjvm.AbstractMasterJVM$1.run(
           AbstractMasterJVM.java:197)

Testcase: testInterpretCurrentInteractionWithIncompleteInput(edu.
   rice.cs.drjava.model.repl.InteractionsModelTest):      Caused
   an ERROR
The test did not perform a join on all spawned threads.


Thread Thread-2 (edu.rice.cs.util.
   ReaderWriterLockTest$PrinterReaderThread) is still alive:
   state=TERMINATED

Testcase: testMultipleReaders(edu.rice.cs.util.
   ReaderWriterLockTest):   Caused an ERROR
The test did not perform a join on all spawned threads.
```

Listing 2.5: Concurrency Problems in DrJava Detected by Improved JUnit

Chapter 3

Execution under Various Schedules

The previous chapter introduced an improvement to JUnit that records failed assertions and uncaught exceptions in all threads, ensures that all child threads have ended, and that those child threads were actually joined to the main thread.

Unfortunately, these checks are only performed in the schedule that happened to execute. A problem could occur in another schedule and go undetected. It is therefore valuable to execute programs using different schedules.

It is not necessary to consider all possible schedules at the instruction level. If all variables that are shared among several threads are either (i) volatile, or (ii) consistently protected by at least one lock when they are accessed, then it is sufficient to change the schedule at the level of *critical blocks*, which are delimited by accesses to shared variables or other actions that influence concurrency [6]. Such accesses to shared variables and concurrency operations are called *synchronization points*.

3.1 Synchronization Points

Synchronization points are all those operations in a program that are susceptible to changes in the schedule, such as reads of shared variables; or that themselves change the concurrent behavior of the program, e.g. by starting a new thread.

The operations that affect the schedule of a Java program can be localized to a small number of methods and bytecode opcodes, shown in table 3.1 (`Thread` class), table 3.2 (`Object` class), table 3.3 (lock operations), table 3.4 (volatile field accesses), table 3.5 (non-volatile field accesses), and table 3.6 (array accesses). A checkmark ✓ in the "Enabled" columns indicates that the synchronization point described in that row was instrumented with a random delay in the experiments described in section 3.5.

Operation	Time	Description	Enabled
		`java.lang.Thread`	
`start()`	before	parent thread creates a child thread	
`start()`	after	parent thread creates a child thread	✓
`run()`	before	child thread begins to execute	
`exit()`	before	thread ends	
`join()`	before	current thread waits for another thread to end	✓

Table 3.1 : Synchronization Points in `java.lang.Thread`: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Operation	Time	Description	Enabled
		`java.lang.Object`	
`notify()`	before	current thread sends notification to one thread	✓
`notify()`	after	current thread sends notification to one thread	
`notifyAll()`	before	current thread sends notification to all threads	✓
`notifyAll()`	after	current thread sends notification to all threads	
`wait()`	before	current thread waits for notification	✓
`wait()`	after	current thread has received notification	

Table 3.2 : Synchronization Points in `java.lang.Object`: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Operation	Time	Description	Enabled
Opcodes: Locks			
MONITORENTER	before	current thread attempts to acquire a lock	✓
MONITORENTER	after	current thread has acquired a lock	
MONITOREXIT	before	current thread releases a lock	✓
MONITOREXIT	after	current thread releases a lock	

Table 3.3 : Synchronization Points for Locks: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Operation	Time	Description	Enabled
Opcodes: Volatile Field Access			
GETSTATIC	before	read from a volatile static field	✓
PUTSTATIC	before	write to a volatile static field	✓
GETFIELD	before	read from a volatile non-static field	✓
PUTFIELD	before	write to a volatile non-static field	✓

Table 3.4 : Synchronization Points for Volatile Fields: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Operation	Time	Description	Enabled
Opcodes: Non-volatile Field Access			
GETSTATIC	before	read from a non-volatile static field	✓
PUTSTATIC	before	write to a non-volatile static field	✓
GETFIELD	before	read from a non-volatile non-static field	✓
PUTFIELD	before	write to a non-volatile non-static field	✓

Table 3.5 : Synchronization Points for Non-Volatile Fields: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Operation	Time	Description	Enabled
Opcodes: Array Access			
?ALOAD	before	read from an array	
?ASTORE	before	write to an array	

Table 3.6 : Synchronization Points for Arrays: A list of those operations that affect the concurrent behavior of a program and that can be instrumented using Concutest.

Grouping instructions together as critical blocks significantly reduces the number of schedules. Unfortunately, that number still increases exponentially with the number of concurrent threads and the size of the programs, and programs that go beyond the scope of mere examples still become intractable to test comprehensively.

It is unsettling to give up the guarantee that a tested program will pass its unit test suite under all possible schedules. The cursory tests in previous literature [9, 42] and the extensive experiments performed as part of this thesis suggest that inserting random delays at synchronization points represents an effective alternative to comprehensive testing of all schedules.

3.2 Instrumentation with Random Delays

The strategy of inserting random delays at synchronization points is based on the realization that many schedules will likely exhibit a certain concurrency problem, and that it is therefore not necessary to comprehensively text all schedules. As discussed, the most frequent schedule often prevents a bug from being discovered, but small changes in timing are enough to uncover the problem. Inserting a delay every few times the program encounters a synchronization point is sufficient to force the program into a different schedule.

The program is then run many times, and because the synchronization points at which the program sleeps, as well as the durations of the delays, differ from one execution to the next, it is likely that the program explores different schedules.

The success rate of this approach varies depending on the hardware, the operating system, and the nature of the bug. In the experiments described in this chapter, a bug was always found in at least 30% of the executions, and often much more frequently. In the case of DrJava, where the unit tests take about 10 minutes to execute, we can expect to start noticing problems after running the test suite for just an hour. A dedicated machine acting as continuous build server, e.g. using the Hudson [29] continuous integration software, can improve the probability that problems are found even more, while at the same time simplifying the operation for developers: There is no need to run a separate tool, because the build server continuously updates itself to the latest source code revision, builds the application, and runs the test suite under varying schedules.

3.2.1 Delays for `Thread` Methods

Delays can be inserted before and after calls to `Thread.start()`. Inserting a delay after the call is an effective way of increasing the likelihood that the current parent thread does not continue to execute, but that control passes to some other thread, perhaps the child thread.

Inserting a delay before the call to `Thread.start()` can be used to uncover bugs that assume that a child thread has been started without expressing that requirement in code. The same can be accomplished with a delay at the beginning of `Thread.run()`, which is executed when a thread begins to run. Given these two equivalent ways, in the experiments in section 3.5 I chose to insert delays only be-

fore `Thread.start()` and in `Thread.run()`, but not after `Thread.start()`. This choice creates a balance of delays inserted in the parent and the child thread instead of putting two consecutive delays in the parent thread.

Delays could be inserted before or after `Thread.join()`, but there is no need to offer both choices. After the call to `Thread.join()`, only the caller thread continues to execute, and an inserted sleep placed before or after the call will delay the execution of that thread. A delay before `Thread.join()`, however, increases potential concurrency in the sense that more threads are still alive, which is why I chose this location rather than inserting delays after `Thread.join()`. The same result could be achieved with a delay inserted in `Thread.exit()`, which gets executed before a thread terminates; however, placing a delay both in `Thread.join()` and `Thread.exit()` is not necessary.

The methods in the `Thread` class are simple to modify using local instrumentation (see section 6.2): All the changes can be directly inserted into the bytecode of the methods.

3.2.2 Delays for `Object` Methods

The methods in the `Object` class are more difficult to modify. The `notify()`, `notifyAll()`, and `wait()` methods are native and therefore contain no bytecode that can be modified directly. Because of the way the JVM links native methods to their native code, these methods also cannot be renamed and called from wrapper methods that take their places.

The only way to instrument calls to these method is through global instrumentation (see section 6.2): A call to `SyncPointBuffer.randomDelay()`, the method

that implements the random delay, is inserted directly before or after all calls to `notify()`, `notifyAll()`, and `wait()`.

Random delays before `notify()` and `notifyAll()` can change the thread that receives the notification, which can lead to large changes in the schedule. Delays after calls to these methods expose problematic programs that rely on the fact that most schedulers allow the current thread to continue, even after another thread has been notified, without expressing that requirement in code.

Another common class of bugs is the "missed notification." Java does not record the fact that an object has been notified, and if a thread is not waiting for a notification when it is sent, the signal is lost. Because of this, as well as spurious wake-ups*, Java programs should also use a volatile flag and call `wait()` in a loop. Inserting a delay before `wait()` increases the likelihood that a program that does not use a flag will miss a notification.

3.2.3 Delays for Lock Operations

The Java opcodes for acquiring and releasing locks, `MONITORENTER` and `MONITOREXIT`, as well as calls to synchronized methods, also represent synchronization points that should be the targets of random delays.

Random delays inserted before a `MONITORENTER` can change the thread that acquires the lock, which can lead to large changes in the schedule. The possibility of inserting method calls after a `MONITORENTER` and before and after a `MONITOREXIT` largely exists to support other tools, like a schedule recorder and a deadlock monitor [34].

---

*`join()` and `wait()` may resume spuriously (§17.8.1 JLS [43]).

Acquiring or releasing a lock is not performed using a method call; therefore, it is impossible to modify just one method and see the desired change everywhere, which could be done for the methods in the `Thread` class. Just like for the methods in `Object`, calls to `SyncPointBuffer.randomDelay()` have to be inserted directly before and after the `MONITORENTER` and `MONITOREXIT` opcodes.

Synchronized methods are different. There are no opcodes that correspond to the synchronized method acquiring or releasing a lock; instead, this is done implicitly by the JVM. Naively, a call to a synchronized method like the one in listing 3.1 can be instrumented by inserting calls to `SyncPointBuffer.randomDelay()` both at the call site and inside the method, as shown in listing 3.2. Instead, Concutest transforms the synchronized method into an unsynchronized method that contains a synchronized block; the framework then treats the `MONITORENTER` and `MONITOREXIT` opcodes of that synchronized block as described above. This transformation turns what would otherwise be an expensive global instrumentation into a local change.

```
// call site                                                    1
someObject.syncMethod();                                        2
                                                                3
// ...                                                          4
class SomeObject {                                              5
  public synchronized void syncMethod() {                       6
    // ...                                                      7
  }                                                             8
}                                                               9
```

Listing 3.1: Synchronized Method before Instrumentation

```
// call site                                                    1
randomDelay(); // before acquiring lock                         2
someObject.syncMethod();                                        3
randomDelay(); // after releasing lock                          4
                                                                5
// ...                                                          6
class SomeObject {                                              7
  public synchronized void syncMethod() {                       8
    randomDelay(); // after acquiring lock                      9
    // ...                                                      10
    randomDelay(); // before releasing lock                     11
  }                                                             12
}                                                               13
```

Listing 3.2: Synchronized Method Naively Instrumented

3.2.4 Delays for Field Accesses

Just like lock operations, field accesses are performed using opcodes. Consequently, adding delays before field accesses requires a global instrumentation of all methods that read or write a field. The opcodes that correspond to read and write operations are `GETSTATIC` and `PUTSTATIC` for static fields and `GETFIELD` and `PUTFIELD` for non-static fields.

In a race-free program, shared fields are either volatile, or all threads accessing a shared field consistently hold at least one lock when reading from or writing to that field. In that case, it is not necessary to add delays before accessing non-volatile fields because the field access is on the inside of a critical block, and a delay is already inserted when the lock is acquired.

To minimize unnecessary delays, Concutest determines at instrumentation time whether an accessed field is volatile and only inserts delays if that is the case. Unlike the `static` modifier, which causes the compiler to emit `GETSTATIC` and `PUTSTATIC` opcodes instead of `GETFIELD` and `PUTFIELD` opcodes, the `volatile`

```
someObject.syncMethod();

// ...
class SomeObject {
  public void syncMethod() { // not synchronized anymore
    randomDelay(); // before acquiring lock
    synchronized(this) {
      randomDelay(); // after acquiring lock
      // ...
      randomDelay(); // before releasing lock
    }
    randomDelay(); // after releasing lock
  }
}
```

Listing 3.3: Synchronized Method Instrumented as Method with Synchronized Block

modifier has no impact on the generated opcode. Finding out whether a field is volatile is therefore more involved and requires finding the class that contains the field on the classpath, loading it, and examining its list of fields.

The developer can also choose to add delays before non-volatile variable accesses as well. Doing so increases the number of delays that are inserted, but it may make it unnecessary to run a race detector in parallel.

3.2.5 Delays for Array Accesses

The JVM implements array accesses using the `?ALOAD` and `?ASTORE` families of opcodes (e.g. `IALOAD` and `IASTORE` to read and write integer arrays, respectively). The instrumentation is similar to that of field accesses described above.

Since only the array object itself can be declared volatile, but not the array elements[†], accesses to all array elements have to be treated as synchronization points.

[†]Using Type Annotations (JSR 308) [47], a `@Volatile` annotation could be placed on the array dimension: `int @Volatile[] i;` could be a non-volatile field with volatile `int` array elements.

Depending on the compiler, an exception can be made if the array object has been loaded from a field. The javac compiler, for example, emits fresh `GETFIELD` and `PUTFIELD` or `GETSTATIC` and `PUTSTATIC` opcodes every time an array stored in a field is accessed. javac does not cache the value of the array object on the stack by loading and then duplicating it using `DUP`, even if the two accesses are very close to each other and the field storing the array is final. Because of the close proximity of the opcode that loads the array from the field and the opcode that reads or writes one of the array elements, it is unnecessary to add an additional delay between those two operations.

This optimization only applies to javac-generated code, and it does not hold for arrays in local variables. If an array in a local variable could be shared across thread boundaries, then all `?ALOAD` and `?ASTORE` opcodes operating on the array should be instrumented with random delays.

3.3 Instrumentation with Random Yields

An instrumentation strategy that inserts random yields instead of random delays has also been developed. Instead of inserting a call to `Thread.sleep()` that gets executed with a certain probability, a similar call to `Thread.yield()` is inserted.

This strategy is conceptually simpler than the strategy that uses delays:

- The call to `Thread.yield()` is almost a no-op if the program is not multi-threaded and there is no other thread that can execute. The delay strategy, described above, maintains a count of the running threads to avoid inserting delays in single-threaded situations.

- The parameter space is smaller for `Thread.yield()` than for `Thread.sleep()`. For the latter, the tester needs to choose the probability that a delay is executed at all, and then describe the distribution of the durations. For `Thread.yield()`, the only parameter is the probability that it is executed.

The effectiveness of the strategy that uses yields has not been thoroughly researched, though. Previous studies indicate that random yields are less effective than random delays [9, 42].

3.4 Restrictions on Programs

For the strategy of inserting random delays at synchronization points to be effective, the program has to fulfill several requirements:

- All uncaught exceptions and failed assertions have to be detected, regardless of the thread in which they occur.
- For the unit testing framework to function properly, all child threads should be joined by the test's main thread.

The simplest way of ensuring the previous two requirements is to use an improved unit testing framework such as ConcJUnit, introduced in the previous chapter.

- The program must employ a consistent locking discipline when accessing data that is shared across threads. In the case of Java, this requires that a shared variable is either volatile, or that at least one lock is consistently held every time that variable is accessed.

This third requirement essentially implies that the program is race-free, as defined by the Java Memory Model [46]. The "races" occur only when attempting to acquire locks or when accessing volatile variables. To ensure that a program meets this third requirement, the programmer can run a race detector such as FastTrack [12] in parallel with Concutest.

These requirements do not seriously impede programmers, and without meeting them, Concutest may not be effective at detecting most concurrency problems.

It may also be useful for the programmer to manually specify which volatile variables and which arrays are being shared between threads, and which classes should be instrumented. There are numerous volatile variables and arrays in the Java API, and many of them may not have an effect on the concurrent behavior of the program.

In fact, many of the volatile variables and arrays in a program may not even be shared. It is possible to use static analysis to conservatively determine which variables and arrays could be shared, for example using Soot's "may happen in parallel" (MHP) [39, 24] analysis.

Unfortunately, Soot's MHP implementation does not scale to production-size programs. At this time, instrumenting all user classes, but not most of the Java API, is a viable compromise.

3.5 Results

To test the effectiveness of running unit tests under varying schedules by inserting random delays or yields, I created several tests representative of common concur-

rency problems and established whether the instrumentation helped in detecting the problems.

Several of the examples are directly inspired by the tests used in the literature to evaluate the effectiveness of ConTest [9] and rstest [42]. Unfortunately, direct quantitative comparisons are not possible, since neither ConTest nor rstest made source code or information about the test hardware available. The tests were described in prose, though, and independently re-implemented. Concutest, the framework developed as part of this thesis, was able to detect all the concurrency problems that ConTest or rstest were able to detect.

The results are described in the sections below and summarized in the following tables. Table 3.7 provides the percentages of the errors detected by the two previous frameworks, ConTest and rstest, and the average percentage of errors detected by Concutest. Table 3.8 shows the number of experiments run with ConTest and rstest, and the average number of experiments run with Concutest. In most cases, the performance of Concutest exceeds that of ConTest, and Concutest is at least comparable to rstest.

3.5.1 Configurations

Note that both the percentage of errors detected and the number of experiments in tables 3.7 and 3.8 are averages for Concutest, since Concutest was tested on four different hardware and software configurations:

- **i7** configuration: Apple MacBook Pro (6.2), Intel Core i7, 2.66 GHz, 4 GB RAM, Mac OS X 10.6.6

- **Core Duo** configuration: Apple MacBook (1.1), Intel Core Duo, 2 GHz, 2 GB RAM, Mac OS X 10.4.11

- **Core 2 Duo** configuration: Dell Dimension 9200, Intel Core 2 Duo, 2.4 GHz, 4 GB RAM, Windows XP

- **i7 Quad** configuration: Dell Studio 435MT, Intel Core i7, 2.66 GHz, quad core, 4 GB RAM, RedHat Enterprise Linux RHEL 4.1.2-46

Tables 3.9 and 3.10 break the results down and show the percentages of errors detected and the numbers of experiments run, respectively, for the different configurations.

It is noteworthy that the experiments were run many more times than ConTest's and rstest's experiments [9, 42]. This was one of the measures undertaken to ensure reproducibility, along with using Concutest on multiple configurations and publishing the source code for the experiments.

Experiment	ConTest			rstest		Concutest	
Errors Detected	none	sleep	yield	none	sleep	none	sleep
ConTestOne	0.0%	20.0%	0.3%			0.6%	33.0%
FundManagers	0.0%	80.0%		0.0%	100.0%	13.8%	100.0%
ConTestThree	0.0%	35.0%				0.2%	79.7%
ConTestFour	0.0%	?				63.1%	99.8%
RSTestOne				0.0%	100.0%	9.3%	99.8%

Table 3.7 : Summary of Scheduling Experiments: Percentage of Errors Detected

Experiment	ConTest			rstest		Concutest average	
Experiments Run	none	sleep	yield	none	sleep	none	sleep
ConTestOne	1000	1000	1000			14644	12483
FundManagers	1000	1000		10 min?	10?	35385	10525
ConTestThree	500	2000				12237	12208
ConTestFour	1000	1000				13489	11907
RSTestOne				10?	10?	11898	11958

Table 3.8 : Summary of Scheduling Experiments: Number of Experiments Run

Experiment Errors Detected		Concutest i7	Concutest Core Duo	Concutest Core 2 Duo	Concutest i7 Quad	Concutest average
ConTestOne	none	0.0%	0.3%	1.4%	0.7%	0.6%
	sleep	41.0%	29.6%	31.1%	30.3%	33.0%
FundManagers	none	8.3%	0.1%	3.6%	43.2%	13.8%
	sleep	99.9%	100.0%	100.0%	99.9%	100.0%
ConTestThree	none	0.0%	0.0%	0.0%	0.7%	0.2%
	sleep	96.2%	96.2%	96.3%	30.3%	79.7%
ConTestFour	none	70.7%	2.3%	79.6%	99.9%	63.1%
	sleep	99.8%	99.5%	100.0%	100.0%	99.8%
RSTestOne	none	0.7%	1.0%	35.2%	0.4%	9.3%
	sleep	99.7%	99.8%	99.8%	99.8%	99.8%

Table 3.9 : Detailed Results for Scheduling Experiments: Percentage of Errors Detected

3.5.2 Test Parameters

For all of these experiments, random delays with durations of 75 to 150 ms, linearly distributed, were inserted with a probability of 0.4 when more than one thread was running.

The enabled insertion sites were:

Experiment	Experiments Run	Concutest i7	Concutest Core Duo	Concutest Core 2 Duo	Concutest i7 Quad	Concutest average
ConTestOne	none	10001	12574	26000	10000	14644
	sleep	12356	15494	12080	10000	12483
FundManagers	none	52200	30300	27000	32041	35385
	sleep	10800	10500	10800	10000	10525
ConTestThree	none	10628	11278	17041	10000	12237
	sleep	11355	15394	12081	10000	12208
ConTestFour	none	10528	11169	22259	10000	13489
	sleep	11255	15294	11080	10000	11907
RSTestOne	none	10628	11267	15697	10000	11898
	sleep	11356	15394	11081	10000	11958

Table 3.10 : Detailed Results for Scheduling Experiments: Number of Experiments Run

- after `Thread.start()` (in the parent thread),
- before `Thread.join()`,
- before `Object.notify()` and `Object.notifyAll()`,
- before `Object.wait()`,
- before acquiring a lock, i.e. before `MONITORENTER`, and
- before accessing a volatile field.

This subset of enabled insertion sites was chosen to increase the probability of encountering common concurrency problems:

- By delaying the parent thread after `Thread.start()`, it becomes more likely that the child thread starts executing immediately, a behavior that is usually uncommon.

- Delaying a thread before or after `Thread.join()` challenges the assumption that a thread which joins a dead thread continues executing immediately.

- Inserting delays before `Object.notify()` and `Object.notifyAll()` can change the threads that receive the notification.

- Delaying a call to `Object.wait()` makes a "missed notification" problem more likely, in which a Java notification is lost because no other thread is waiting for it.

- A delay before acquiring a lock changes the order in which threads execute synchronized blocks.

- Delaying accesses to volatile fields can expose atomicity violations.

The experiments exhibiting some of these common concurrency problems are described below.

3.5.3 Experiment 1: Race

The first test, inspired by ConTest's first experiment and shown in the appendix in listing B.1, has several threads racing to set a flag first. The threads are normally too short to be preempted, and each thread usually finishes before the next thread even starts. This is, however, not ensured using program constructs; therefore, it is possible for several threads to claim to be "first" if a context switch occurs after the flag is checked in line 9 and before the flag is set in line 10.

According to the Edelstein paper [9], ConTest detected the bug in 200 of 1000 runs when the program was seeded with random sleeps, and in 3 out of 1000 runs

when random yields were used. Without sleeps or yields, the bug was never observed.

This experiment was run at least 10000 times on each of the four configurations without Concutest, and no 0.0%, 0.3%, 1.4%, and 0.7% of the errors were detected on i7, Core Duo, Core 2 Duo, and i7 Quad, respectively. Observing this kind of race condition is therefore very rare. Concutest then added random sleeps to the program, and the experiment was repeated over 12000 times on each platform. The detection rate jumped remarkably to 41.0% on i7, 29.6% on Core Duo, 31.1% on Core 2 Duo, and 30.3% on i7 Quad. These detection rates exceed the published detection rate of ConTest [9].

Note that ConTest was able to detect this problem by itself when all variable accesses were instrumented with delays or yields. The race detector that should run in parallel with Concutest, though, would have alerted the programmer to the presence of a race condition. With synchronized blocks inserted, the variable changed to `volatile`, or delays specifically enabled for the variable, Concutest was also able to expose the non-determinism.

3.5.4 Experiment 2: Atomicity

The second test, used by ConTest as well as rstest, was first presented in a JDC Tech Tip [48]. The Tech Tip archive was not available anymore, but the source the ConTest example was based on was eventually retrieved. The salient parts of the source code are shown in listing B.2 and listing B.3.

In this experiment, a simulation of a stock exchange, several threads perform transfers from one account to another. In doing so, they add or subtract from shared data without performing synchronization to ensure that the reads and writes

are atomic: The read-and-write operation `balances[t.fundFrom] -= t.amount;` in line 5 (and similarly `balances[t.fundFrom] -= t.amount;` in line 6) generates the non-atomic bytecode instructions in listing 3.4.

```
GETSTATIC balances                                                    1
ALOAD_0                                                               2
GETFIELD fundFrom                                                     3
DUP2                                                                  4
IALOAD                                                                5
ALOAD_0                                                               6
GETFIELD amount                                                       7
ISUB                                                                  8
IASTORE                                                               9
```

Listing 3.4: Annotated Classes

If a context switch occurs after a thread has executed the `IALOAD`, but before it has executed the `IASTORE`, and another thread also begins to execute the same read-and-write operation, one of the writes may be lost.

ConTest detected the bug in 800 of 1000 runs when the program was seeded with random sleeps, but never observed without the inserted sleeps. Since the ConTest paper mentions that Edelstein et al. "modified the original program so that `move_money` contained busywork (a loop of empty writes)" [9], but does not describe the modifications in closer detail, it is even more difficult to quantitatively compare the effectiveness of Concutest to that of ConTest.

In rstest's version without sleeps or yields, the bug was never observed in 10 runs with "few thousand transfers." When rstest inserted sleeps or yields, the bug "manifested itself many times in each execution." [42] The way these results are stated makes a quantitative comparison impossible.

The experiment was run over 27000 times on each of the four configurations, and without Concutest, the defect was detected 8.3%, 0.1%, 3.6%, and 43.2% of the times on i7, Core Duo, Core 2 Duo, and i7 Quad, respectively. Reliably detecting this kind of problem without help is problematic, except perhaps on the quad-core i7, which can run more threads in concurrently.

After Concutest had inserted random sleeps into the program, the experiment was repeated at least 10000 times on each configuration, and the detection rate jumped to nearly 100%. Concutest makes atomicity violations trivial to observe, and its detection rates exceed that of ConTest [9] and are comparable to that of rstest [42].

3.5.5 Experiment 3: Uninitialized Data

ConTest also provided the inspiration for the third test, shown in listing B.4. The most important pieces of this example were available in the ConTest paper [9], but details about how these parts were set up were missing again.

Again, there was a problem with atomicity, as well as a race condition. If a context switch occurs after the `notified = true;` in line 32, but before the `subject` field can be initialized in the following line, the thread may use the uninitialized `subject` field in line 29.

The ConTest paper states that the bug was never observed in any of 500 runs, each with four threads. When instrumented with random sleeps or yields, ConTest revealed the problem "about 700 times in 2000 tests" [9].

This experiment was run at least 10000 times on each of the configurations without the help of Concutest, and the error was never detected on i7 (0.0%), once on Core Duo (0.0%), eight times on Core 2 Duo (0.0%), and 71 times on i7 Quad

(0.7%). After inserting random sleeps using Concutest and repeating the experiment at least 10000 times on each platform, 96.2%, 96.2%, 96.3%, and 30.3% of the defects were observed on i7, Core Duo, Core 2 Duo, and i7 Quad, respectively. Except for the i7 Quad detection rate, these rates far exceed the published detection rate of ConTest. The i7 Quad detection rate is similar to the rates published for ConTest [9].

3.5.6 Experiment 4: Chain of Threads

The fourth test is the final example that was provided by ConTest. It is shown in listing B.5.

Ten threads are spawned recursively in line 29 under the assumption that the current thread continues to execute; if, however, the spawned child thread starts executing immediately, it will attempt to get a value from a hash map that has not been inserted by the parent thread in line 30 yet, causing a `NullPointerException` in line 26.

As part of the ConTest study, the experiment was repeated 1000 times. The results (370 faults for NT, 12 for yield, 0 for sleep, and 6 for priority [9]), however, are not well explained. Instead, Edelstein et al. focus on how instrumentation with ConTest changes the number of concurrent threads in the experiment, and interpret the results as evidence that ConTest can lessen the impact of the operating system on program behavior.

As part of this thesis, the experiment was run at least 10000 times on each configuration without instrumentation by Concutest. On the Core Duo, the least powerful of the configurations, the problem was apparent only 2.3% of the time. On the Core 2 Duo, the i7, and the i7 Quad, the defect could be detected in 79.6%, 70.7%,

and 99.9% of the executions without the help of Concutest. This demonstrates that some problems that hardly ever appear on less powerful machines may become very obvious on more recent systems.

When Concutest added bytecode for random delays and the experiment was re-run over 11000 times, detection rates increased to 99.8% on the i7, 99.5% on the Core Duo, 100.0% on the Core 2 Duo (11079 out of 11080 defects detected), and 100.0% on the i7 Quad (9994 out of 10000 defects detected).

3.5.7 Experiment 5: Missed Notification

The fifth test was used as first example in the rstest paper. It is based on NASA's Remote Agent, a spacecraft controller developed at NASA Ames Research Center. The source code of the example, shown in listing B.6, was re-constructed using an excerpt found in a Java PathFinder paper [54] that was cited in the rstest paper [42].

The problem in NASA's Remote Agent is a lack of atomicity. If a context switch occurs in line 38 after the conditional has evaluated to `true` but before the resulting `event1.wait_for_event();` in line 39 can be executed, then it is possible for the thread to miss the notification sent by `notifyAll()` when another thread invokes `e1.signal_event();`. As a result, the first thread now waits forever.

Without rstest, no deadlock was observed "in 10 minutes of execution." When using rstest in 10 runs, a deadlock on average occurred after just 0.5 seconds.

When run at least 10000 times without the help of Concutest on each configuration, the deadlock was observed only 0.7% and 1.0% of the times on the Mac OS X-based i7 and Core Duo configurations and 0.4% of the time on the Linux-based i7 Quad, but 35.2% of the times on the Windows-based Core 2 Duo configuration.

This shows that the operating system may have a major impact on the visibility of a bug as well.

After Concutest had added random delays in critical places and the program was run again over 11000 times per configuration, the detection rates were nearly uniform at 99.7%, 99.8%, 99.8%, and 99.8% for i7, Core Duo, Core 2 Duo, and i7 Quad, respectively. Concutest can therefore be used to temper the impact of the operating system choice on testing.

The results of all of these experiments show that Concutest is a powerful tool for testing concurrent programs under varying schedules.

Chapter 4

Execution Logging

The previous two chapters presented necessary tools for testing concurrent programs. Without a unit testing framework that detects problems in all threads and a component that allows execution under various schedules, it is likely that concurrency bugs are missed during testing.

Writing unit tests for concurrent programs is nonetheless difficult. This chapter and the next chapter focus on making that task simpler. The execution logger described in this chapter records whether certain parts of the code were executed, and if so, how often. These pieces of information can then be checked in the unit tests.

Test and application source code should remain loosely coupled. Ideally, there should be no burden on the performance or complexity of the application source code just to allow testing. Unfortunately, testing becomes difficult when methods do not have a return value that can be tested directly. Graphical user interface (GUI) frameworks and thread-related parts of the Java API often employ the `Runnable` interface shown in listing 4.1, where exactly that is the case: The `run()` method does not return a value, and whether the method executed can only be established by looking for side effects, which may be difficult to observe.

The traditional approach to make `void` methods testable is to introduce a flag that is set by the method in question when it is executing. Doing so introduces

```
package java.lang;                                    1
public interface Runnable {                           2
  public void run();                                  3
}                                                     4
```

Listing 4.1: Runnable Interface

testing concerns into the application that do not belong there and that could potentially be costly if the method being logged is executed frequently.

Using Java annotations, the programmer can encode which methods should be logged as metadata—data about the program—without changing the application source code. A program then processes the annotations using bytecode rewriting to introduce logging only in those methods that require it, and only in test builds. The released application code does not contain any logging instructions at all.

4.1 Logging Annotations

The execution logging tool introduced as part of this thesis provides several ways of specifying which methods should be logged.

4.1.1 Using the `@LogThis` Annotation

The simplest way to indicate which methods should be logged is by placing a `@LogThis` annotation directly in front of the method that should be logged. To log all methods in a class, `@LogThis` can also annotate a class. Examples of both are shown in listing 4.2. A unit test can then assert that a method was or was not executed or compare the execution count. A unit test like that is shown in listing 4.3.

The first `@LogThis` annotation in line 2 of listing 4.2 causes the `noResult()` method of the `LogJustThisMethod` class to be logged; the `notLogged()` method

```
class LogJustThisMethod {
  @LogThis
  public void noResult() { // no return value
    // do something
  }

  public void notLogged() {
    // not logged
  }
}

@LogThis
class LogAllMethods {
  public void alsoNoResult(double d) { // no return value
    // do something
  }

  // has return value, but is also logged
  public int timesTwo(int i) {
    // do something
    return i*2;
  }
}
```

Listing 4.2: Examples of Using `@LogThis`

```
import static edu.rice.cs.cunit.execLog.ExecutionLog.*;

public class LogThisTest extends TestCase {
  public void testLog() {
    assertHasNotExecuted("LogJustThisMethod.noResult()");
    assertHasNotExecuted(LogAllMethods.class, "alsoNoResult(
       double)");
    assertHasNotExecuted("LogAllMethods.timesTwo");
    new LogJustThisMethod().noResult();
    LogAllMethods o = new LogAllMethods();
    o.alsoNoResult(3.14);
    int i = o.timesTwo(3);
    int j = o.timesTwo(5);
    assertHasExecuted("LogJustThisMethod.noResult()");
    assertHasExecuted("LogAllMethods.alsoNoResult(double)");
    assertHasExecuted(LogAllMethods.class, "timesTwo");
    assertExecutionCountEquals("LogAllMethods.timesTwo", 2);
  }
}
```

Listing 4.3: Unit Test Referring to Methods and Classes Annotated with `@LogThis`

is not logged. The second `@LogThis` annotation in line 12 enables logging for all methods in the `LogAllMethods` class of the example.

Lines 5 to 7 in listing 4.3 then assert that none of the three logged methods have been executed at the beginning of the test. There are two different ways of specifying the method, either as combined class and method string, or using a class literal and a method string*. Note that providing the parameter types is only necessary if the method is overloaded; if the name of a method uniquely identifies it, the parentheses and parameter types can be omitted, as shown in line 7

In lines 8 to 12, the test makes several calls to the methods being logged. Lines 13 to 15 then assert that the methods have indeed been executed. The last line of the test method compares the actual execution count of the `timesTwo` method in the `LogAllMethods` class to the expected count of 2. It is an error for a unit test to make assertions about methods that have not been logged, or to use ambiguous method strings.

4.1.2 Using the `@LogTheMethod` and `@LogTheClass` Annotations

Specifying methods and classes that should be logged using `@LogThis` annotations makes it unnecessary to introduce a field and add code to set that field; however, it is still necessary to change the source code of the application.

---

*A *method literal*, similar to a `MyClass.class` class literal, would be useful. For example, `MyClass.myMethod.method` could be used to refer to the `myMethod()` method in the `MyClass` class. To deal with overloaded methods, parameter types would have to be specified, as in `MyClass.myMethod.method(int)`. The type of the method literal should be the appropriate `java.lang.reflect.Method` instance.

```
import static edu.rice.cs.cunit.execLog.ExecutionLog.*;

public class LogTheXTest extends TestCase {
  @LogTheClass({LogAllMethods.class})
  @LogTheMethod({
    @MethodDesc("LogJustThisMethod.noResult()")
  })
  public void testLog() {
    assertHasNotExecuted("LogJustThisMethod.noResult()");
    // ...
    assertExecutionCountEquals("LogAllMethods.timesTwo", 2);
  }
}
```

Listing 4.4: Unit Test Referring to Methods and Classes using `@LogTheClass` and `@LogTheMethod`

Using the `@LogTheClass` and `@LogTheMethod` annotations, the programmer can specify in the test code which methods should be instrumented for logging. The two annotations in listing 4.4 accomplish the same as the two `@LogThis` annotations shown above in listing 4.2, making it unnecessary to change the application code for testing.

Note that the arguments to the `@LogTheClass` and `@LogTheMethod` annotations are both arrays. This is necessary because Java does not allow multiple occurrences of the same annotation in the same place. Java does not allow the syntax

```
@LogTheClass(Foo.class)
@LogTheClass(Bar.class)
void m() { ... }
```

and instead requires an array as alternative:

```
@LogTheClass({Foo.class, Bar.class})
void m() { ... }
```

The change above is relatively simple for the `@LogTheClass` annotation, but it becomes more difficult for `@LogTheMethod`, which should provide several ways of specifying the method, e.g. as a combined class and method string, or as class literal and method string. Here, the desired syntax

```
@LogTheMethod("Foo.someMethod")
@LogTheMethod(c=Bar.class, m="otherMethod")
void m() { ... }
```

has to be replaced by a much more cumbersome representation that uses a helper annotation for method descriptions, `@MethodDesc`:

```
@LogTheMethod({
  @MethodDesc("Foo.someMethod"),
  @MethodDesc(c=Bar.class, m="otherMethod")
})
void m() { ... }
```

Another advantage of providing the set of methods for which execution should be logged at the beginning of a test method is that logging can be enabled for that test only. Other tests, which do not need these methods to be logged, will not be impacted by the instrumentation. Such a differentiation, implemented using a custom class loader, allows execution logging of methods for some basic unit tests without slowing down the execution of aggregate unit tests that may call those methods frequently.

4.1.3 Logging Anonymous Inner Classes

Using `@LogTheMethod` and `@LogTheClass` annotations to specify which methods should be logged for a certain test works well. It keeps the application code completely free of any test-related concerns.

Unfortunately, the methods to be logged are referred to by name, which becomes problematic with anonymous inner classes that do not have a name.

There are several ways of still identifying a method in an anonymous inner class exactly:

- Use the anonymous inner class number, e.g. `SomeClass$1.someMethod()`. This solution is brittle: The number depends on the order of anonymous inner classes in the containing class. Furthermore, the number is chosen by the javac compiler, and the numbering scheme could change from one version to the next.

- Use a line number to identify the method. This choice is also brittle, as it depends on the layout of the enclosing source code.

- Use `@LogThis` in the application code. This solution precisely identifies the method of interest, but introduces testing concerns into the application source code, as explained above.

- Extend the `@LogTheMethod` and `@LogTheClass` annotations to allow logging in methods of subclasses to be enabled. The annotation can then specify base class that the anonymous inner class extends. An example of this approach is shown in listing 4.5.

```
import static edu.rice.cs.cunit.execLog.ExecutionLog.*;

public class LogTheXAnonymousTest extends TestCase {
  @LogTheMethod({
    @MethodDesc(value="Runnable.run()", subclasses=true)
  })
  public void testLog() {
    assertHasNotExecuted("Runnable.run()");
    LogClassWithAIC().doSomething();
    // execution count is 2, not specific to any
    // Runnable subclass
    assertExecutionCountEquals("Runnable.run()", 2);
  }
}
```

Listing 4.5: Example of Using `@LogTheMethod` with Anonymous Inner Classes

The latter solution, enabling logging in subclasses, is the most robust of the choices. It keeps test concerns completely out of the application code. This approach, however, lacks specificity. The final execution count for `Runnable.run()` in listing 4.5 after executing the code in listing 4.6 is 2, because the logging system does not distinguish between the two subclasses of `Runnable`.

It is possible to extend the `@MethodDesc` method description annotation to include more detail to reclaim the specificity, e.g. by mentioning the enclosing method by name, as shown in listing 4.7. Other parameters to make method descriptions more specific include the name of the source file name, the return type, and the exact signature, including return type and parameter names.

Note that the example in listing 4.7 still does not allow the test to distinguish between the execution counts of the two `Runnable` anonymous inner classes. The example determines that the execution count for `Runnable` and its subclasses is 1, because only one anonymous inner class was instrumented for logging. To verify that the methods in both anonymous inner classes were executed once, the pro-

```
public class LogClassWithAIC {
  public static void doSomething() {
    new Runnable() {
      public void run() {
        System.out.println("First AIC");
      }
    }.run();
    somethingElse();
  }
  public static void somethingElse() {
    new Runnable() {
      public void run() {
        System.out.println("Second AIC");
      }
    }.run();
  }
}
```

Listing 4.6: Example Application Code with Anonymous Inner Classes to be Logged Using `@LogTheMethod`

```
import static edu.rice.cs.cunit.execLog.ExecutionLog.*;

public class LogTheXAICEnclosingTest extends TestCase {
  @LogTheMethod({
    @MethodDesc(
      value="Runnable.run()", subclasses=true,
      enclosing="LogClassWithAIC.doSomething"
    )
  })
  public void testLog() {
    assertHasNotExecuted("Runnable.run()");
    LogClassWithAIC().doSomething();
    // execution count is 1, only enclosed in doSomething()
    assertExecutionCountEquals("Runnable.run()", 1);
  }
}
```

Listing 4.7: Example of Specifying the Enclosing Method when Using `@LogTheMethod` with Anonymous Inner Classes

```
import static edu.rice.cs.cunit.execLog.ExecutionLog.*;

public class LogTheXAICFriendlyTest extends TestCase {
  @LogTheMethod({
    @MethodDesc(
      value="Runnable.run()", subclasses=true,
      enclosing="LogClassWithAIC.doSomething",
      friendly="Runnable1"
    )
    @MethodDesc(
      value="Runnable.run()", subclasses=true,
      enclosing="LogClassWithAIC.somethingElse",
      friendly="Runnable2"
    )
  })
  public void testLog() {
    assertHasNotExecuted("Runnable1");
    assertHasNotExecuted("Runnable2");
    LogClassWithAIC().doSomething();
    // execution count is 1 for each friendly name
    assertExecutionCountEquals("Runnable1", 1);
    assertExecutionCountEquals("Runnable2", 1);
  }
}
```

Listing 4.8: Example of Specifying Friendly Names when Using `@LogTheMethod` with Anonymous Inner Classes

grammer can add "friendly names" to the `@LogTheMethod` annotations and use them in the test, which is shown in listing 4.8.

One problem with adding more optional detail to the `@MethodDesc` annotation is that it becomes cumbersome to check whether all necessary parts of an optional component have been provided. For example, the enclosing method could be specified using a single class-and-method string, as in listing 4.7, but it could also be provided using two separate members: a class literal and a string with just the method name (and parameter types). Providing just the class literal should be an error.

Furthermore, the framework developer has to decide whether all provided information should be combined in a conjunction, which is what I did here, or in a disjunction. As I pointed out before, it becomes difficult to express richer languages using expressions if subtyping is not available for annotations [34].

4.1.4 Logging Annotations with Subtyping

When subtyping is available for annotations, it is much simpler to create expression languages using annotations. The examples discussed in this section require a compiler that supports subtyping for annotations, such as the xajavac compiler [32].

The main benefit of subtyping for annotations is that it allows the use of the composite design pattern [13]. As part of the execution logging system, I have defined an abstract `@LogLocation` supertype, and several interfaces that extend that supertype, such as `@TheClass`, `@TheMethod`, `@InFile`, and `@EnclosingMethod`. These annotations are leaves in the metadata structure. The Boolean operators `@And`, `@Or`, and `@Not` make up the composite cases and can contain any annotations that extend `@LogLocation`. Since the Boolean operators are annotations themselves, they can be arbitrarily nested as well.

Listing 4.9 shows how these composite cases can be defined with subtyping available. Listings 4.10 and 4.11 contain a few annotation declarations and a usage example.

4.2 Implementations of Execution Logging

I experimented with several different implementations of the logging system. The initial designs used central `HashMap` data structures that stored the execution

```
@interface And extends LogLocation {
  /** @return log location annotations. */
  LogLocation[] value();
}

@interface Or extends LogLocation {
  /** @return log location annotations. */
  LogLocation[] value();
}

@interface Not extends LogLocation {
  /** @return log location annotation. */
  LogLocation value();
}
```

Listing 4.9: Boolean `@And`, `@Or`, and `@Not` Operators for Logging Locations

```
// common superclass
@interface LogLocation { }

// annotation specifying log location and friendly name
public @interface Log {
    /** @return log location annotation. */
    LogLocation value();
    /** @return friendly name for this location. */
    String friendly() default "";
}
```

Listing 4.10: Annotation Definitions for Logging Annotations with Subtyping

```
@Log(
  @Or({
    @And({
      // subclasses of Runnable that are...
      @SubClassOf(Runnable.class),
      // ...defined in MyApp.java...
      @InFile("MyApp.java"),
      // ...but not enclosed in the MyApp.foo method
      @Not(@EnclosingMethod("MyApp.foo"))
    })
    // or the MyApp.foo method
    @TheMethod("MyApp.foo")
  })
)
void testSomething() { /* ... */ }
```

Listing 4.11: Usage Example for Logging Annotations with Subtyping

counts for the different methods. When I replaced a central data structure with individual fields, I saw a significant performance improvement that made the generated code equivalent to hand-written logging code.

- The *naïve* implementation used a single synchronized map for storing all execution counts. Every time a count was accessed, threads had to compete for a single lock for synchronization. This approach was by far the easiest to implement, but it experienced significant lock contention on the dual-core i7.

- In order to avoid lock contention, I created a *non-blocking* implementation using `NonBlockingHashMap` from the Highly Scalable Java library [8]. By storing `AtomicLong` values as execution counts, I was able to completely avoid locking. This implementation avoids the performance degradation the naïve solution displayed with more than four threads on the dual-core i7.

- A non-blocking map is still incurring overhead compared to a completely unsynchronized map. The *per-thread* strategy attempts to reduce this overhead by providing each thread with an unsynchronized map for counting executions. The maps of all threads are later combined into a complete map. Since each thread has its own map, synchronization on that data structure is unnecessary.

  This approach performs similarly well as the non-blocking solution, probably because I used a `NonBlockingHashMap` as central storage for the per-thread maps. Instead, I should have used bytecode rewriting to store these maps directly in each `Thread` object. This probably would have lead to excellent performance while still being simple to implement.

- After having recognized the central data structure as performance bottleneck, I designed the *fields* strategy to avoid any kind of map and more closely match the code of hand-written logging.

  Using bytecode rewriting, the instrumentation strategy augments the `ExecutionLog` class with a separate `AtomicLong` field for each method that should be logged. Synchronization is not necessary since the `incrementAndGet()` operation is atomic, and there is no lookup at all, except for the field lookup that the JVM has to perform anyway.

  This approach matches the performance of hand-written logging closely, even though all the counts are stored in a central class. The values of the fields can easily be retrieved using reflection.

- Storing all counts in the `ExecutionLog` class creates a (somewhat theoretical) limit for the execution logging system: The JVM specification limits the number of fields per class to 65535, and the maximum constant pool size to 65535 entries limits that number even further [44].

  The *local fields* approach places the fields for the execution counts in the same class as the method to be logged. That distributes the fields across all involved classes and makes it much less likely that a field cannot be added due to a JVM limitation.

  The disadvantage of placing the fields in different classes is that lookup is made more difficult.

A study of the generated bytecode and a series of benchmarks, described in the section below, made it clear that *local fields* expectedly performs the best.

4.3 Results

To study the performance of the different logging implementations, I created two small and one large benchmark. These experiments were run several thousand times on a dual-core i7 MacBook Pro and a quad-core i7 Dell desktop. The number of concurrent threads executing the same portion of code was varied from 1 to 16.

The benchmarks consisted of a tight loop, an outer loop, and a number of DrJava unit tests that had used hand-written execution logging. The tight loop example can be summarized by the two lines in listing 4.12: A no-op method is called many times, and each invocation is logged. The outer loop example was similar, but performed a substantial amount of work in the logged method, namely generating a Gaussian blur of an image. Listing 4.13 shows a summary of the benchmark.

The third benchmark consisted of running the DrJava unit tests with a remodeled `GlobalModelTestCase` class. DrJava installs a listener for a number of asynchronous events, and every time the listener's methods are called, a counter is incremented. The listener could be completely replaced by directly logging the methods of various `EventNotifier` subclasses.

The performance of the different implementations was expressed as slowdown factor, compared to the execution with hand-written logging. A slowdown factor of 2.0 means that the execution time for the code generated by the logging system was twice as long as the time with hand-written logging. Lower slowdown factors are therefore better, and a slowdown factor of 1.0 means that there was no slowdown at all, compared to hand-written code.

For the DrJava unit tests, the generated logging code performed as well as the hand-written code. The unit test suite runs for about 10 minutes, and it was impossible to notice any slowdown. The code performing the logging and the unit tests had been simplified, though.

In the tight loop benchmark, both the fields and the local fields strategy were indistinguishable from a hand-written logging implementation. The non-blocking and the per-thread version incurred a slowdown due to centralized storage, but the factor became smaller with a larger number of concurrent threads. A larger number of concurrent threads leads to some form of contention even with the hand-written logging, which makes the overhead of maintaining a central data structure become less significant.

The naïve implementation exhibited a significant slowdown on the dual-core i7 when four or more threads were running concurrently. Interestingly, this marked slowdown was not observed on the quad-core i7.

Figure 4.1 and figure 4.2 show graphs of the slowdown factors of the different implementations for varying numbers of threads on the dual-core i7 and the quad-core i7, respectively. Table 4.1 and table 4.2 provide the numerical data the graphs are based on.

The local fields strategy generally performs best: There is no discernable slowdown, compared to hand-written logging. The fields implementation with all fields in the `ExecutionLog` class performs almost as well. Compared to no logging, the slowdown ranges from about 4% (factor 1.04, fields and local fields, 1 thread, both i7 and i7 Quad) to no slow down (factor 1.00, fields and local fields, 16 threads, both i7 and i7 Quad).

```
for(i=0; i<N; ++i) { loggedMethod(); }                    1
@LogThis void loggedMethod() {/*no op*/ }                 2
```

Listing 4.12: Tight Loop Logging Benchmark

```
for(i=0; i<N; ++i) { loggedMethod(); }                    1
@LogThis void loggedMethod() {                            2
  for(j=0; i<M; ++j) { gaussianBlur(); }                  3
}                                                         4
```

Listing 4.13: Outer Loop Logging Benchmark

When logging the outer loop, all implementations performed comparably well, both to each other and to the hand-written logging. In fact, the graphs in figure 4.3 and figure 4.4 show rather noisy plots, and the error bars suggest that we cannot pick one of the implementations as having performed the best. The conclusion we can draw from this experiment is that all implementations of the logging framework perform as well as hand-written logging, and that choosing the local fields implementation only becomes important when inner loops are being logged.

Table 4.3 and table 4.4 provide the numerical data for figure 4.3 and figure 4.4. μ is the mean slowdown factor, and σ_M is the standard error.

Considering that the local fields implementation is equivalent to hand-written logging code, the execution logging tool provides a convenient and efficient means to keep application and tests decoupled when testing whether certain parts of an application program have executed.

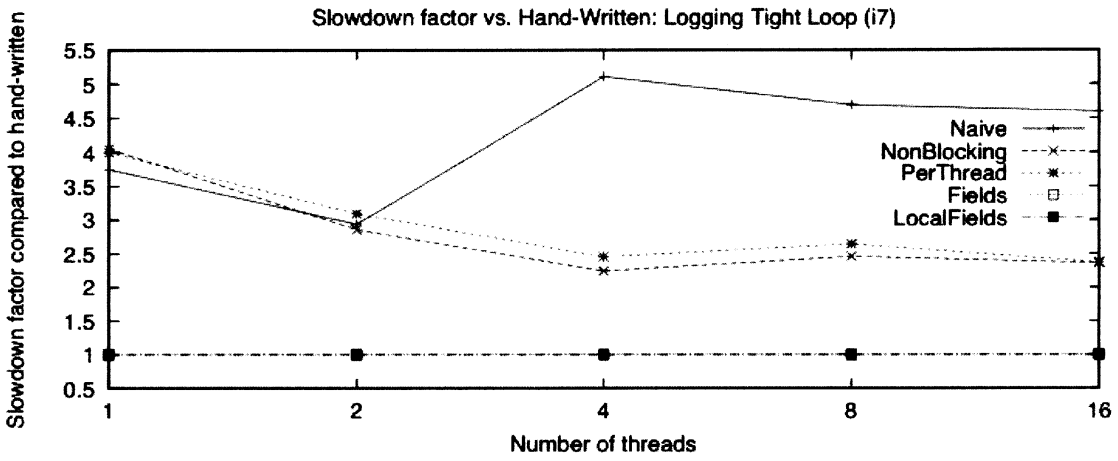

Figure 4.1 : Slowdown Factor for n Threads in a Tight Loop Compared to Hand-written Logging, i7

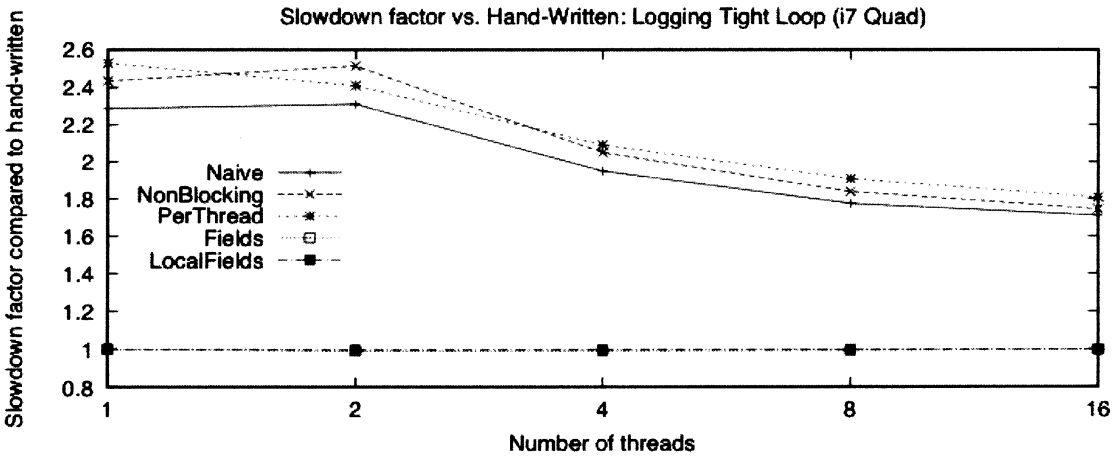

Figure 4.2 : Slowdown Factor for n Threads in a Tight Loop Compared to Hand-written Logging, i7 Quad

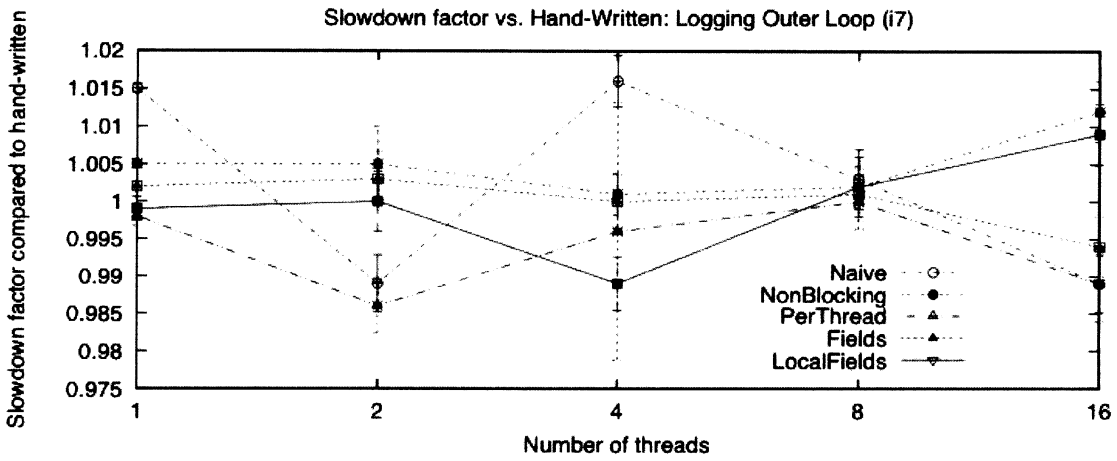

Figure 4.3 : Slowdown Factor for n Threads in an Outer Loop Compared to Hand-written Logging, i7

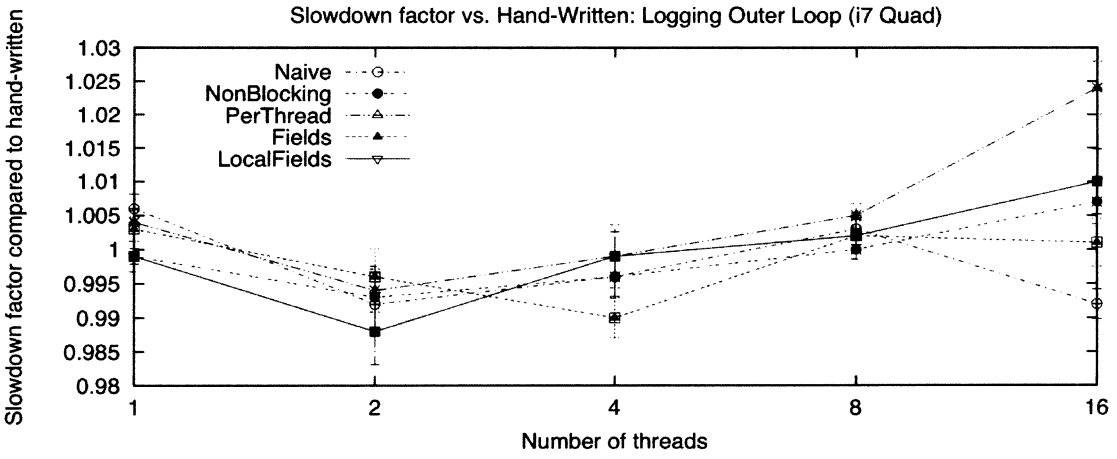

Figure 4.4 : Slowdown Factor for n Threads in an Outer Loop Compared to Hand-written Logging, i7 Quad

Threads	1	2	4	8	16
Naïve	3.748	2.940	5.107	4.690	4.596
NonBlocking	4.054	2.858	2.243	2.458	2.361
PerThread	4.000	3.095	2.455	2.644	2.372
Fields	0.999	0.998	1.003	0.999	1.000
LocalFields	0.998	1.000	0.998	0.998	0.998

Table 4.1 : Slowdown Factor for n Threads in a Tight Loop Compared to Hand-written Logging, i7

Threads	1	2	4	8	16
Naïve	2.287	2.310	1.950	1.774	1.709
NonBlocking	2.434	2.513	2.050	1.839	1.744
PerThread	2.527	2.410	2.091	1.908	1.807
Fields	1.001	0.992	0.993	0.996	0.997
LocalFields	0.999	0.996	0.997	0.997	0.999

Table 4.2 : Slowdown Factor for n Threads in a Tight Loop Compared to Hand-written Logging, i7 Quad

Threads		1	2	4	8	16
Naïve	$\overline{x}$	1.015	0.989	1.016	1.003	0.989
	σ_M	0.000591	0.003881	0.003364	0.003974	0.003894
NonBlocking	$\overline{x}$	1.005	1.005	1.001	1.002	1.012
	σ_M	0.000557	0.004963	0.002743	0.002769	0.003993
PerThread	$\overline{x}$	0.998	0.986	0.996	1.000	0.989
	σ_M	0.001286	0.003697	0.017176	0.003689	0.005066
Fields	$\overline{x}$	1.002	1.003	1.000	1.001	0.994
	σ_M	0.002444	0.003693	0.003618	0.003514	0.004063
LocalFields	$\overline{x}$	0.999	1.000	0.989	1.002	1.009
	σ_M	0.001561	0.004053	0.003620	0.003999	0.004048

Table 4.3 : Slowdown Factor for n Threads in an Outer Loop Compared to Hand-written Logging, i7.

Threads		1	2	4	8	16
Naïve	$\overline{x}$	1.006	0.992	0.996	1.003	0.992
	σ_M	0.002146	0.004578	0.002843	0.002322	0.002181
NonBlocking	$\overline{x}$	0.999	0.993	0.996	1.000	1.007
	σ_M	0.002267	0.004563	0.003140	0.001458	0.003250
PerThread	$\overline{x}$	1.004	0.994	0.999	1.005	1.024
	σ_M	0.001684	0.003112	0.004631	0.001724	0.003969
Fields	$\overline{x}$	1.003	0.996	0.990	1.002	1.001
	σ_M	0.001768	0.004100	0.002955	0.002186	0.003510
LocalFields	$\overline{x}$	0.999	0.988	0.999	1.002	1.010
	σ_M	0.001153	0.004902	0.003581	0.002395	0.004813

Table 4.4 : Slowdown Factor for n Threads in an Outer Loop Compared to Hand-written Logging, i7 Quad

Chapter 5

Invariant Checking

Current development platforms are not designed for concurrency, even though concurrent programs are becoming more prevalent. One of the areas that first became overwhelmingly concurrent was the graphical user interface (GUI) of a program. The previous chapter already discussed how the use of the `Runnable` interface caused a problem for unit testing.

In AWT and Swing, Java's GUI frameworks, the program modifies GUI components and responds to user input by executing `Runnable` objects in a special thread called *event thread* or *event dispatch thread* (EDT). To maintain a responsive user interface, programs performing long computations use both the event thread and a worker thread in the background; therefore, they are concurrent. This serves as an example of how widespread concurrent programs have become.

Along with the problems of testing concurrent programs comes the task of defining, documenting, and enforcing a threading discipline. A threading discipline is defined as a set of rules that dictate which threads must acquire what sets of locks before they may access data.

There are many commonly used examples of these threading disciplines, and a reasonable variety is found in AWT and Swing: For example, the Javadoc documentation states that all the methods defined first in Java's `TreeModel`, `DefaultTreeModel`, `TreeNode`, `MutableTreeNode`, and many other classes associ-

ated with the model side of Swing's tree component may only be called from the event thread. The same applies to classes that belong to Swing's table model, and some methods involved in model-to-view coordinate conversion.

On the other hand, calling `SwingUtilities.invokeAndWait` from the event thread is a recipe for an instantaneous deadlock, since the event thread will wait until the specified task has been completed by the event thread – but the event thread will never even attempt to complete it, because it was told to wait. Both of these limitations define threading disciplines, and these examples merely came from the Swing GUI library; applications and other libraries usually have their own disciplines that need to be followed.

Unfortunately, these threading disciplines are often undocumented, hidden in source code comments, or only found in a white paper about the library. Many times, the authors of concurrent code use several of these approaches to communicate the necessary circumstances for safe access, but there is no way the threading disciplines are enforced. Disobeying a library's threading discipline often does not result in an informative error message, but instead goes unnoticed until much later, when the code has grown and changed, clouding the actual cause of the problem.

This chapter specifies a light-weight language, using Java annotations with subtyping, that allows threading invariants to be applied to methods which require the caller or subclass to adhere to a threading discipline. The invariants are directed outward: They specify contracts that code using the annotated methods must uphold. Most of the time, the verification of adherence is done at runtime, but some verification can be done statically at compile time.

Both the verification of complicated threading invariants at runtime, and the limited static analysis at compile time offer a great benefit to library developers

and users: A library developer can precisely describe the required threading invariants, not in comments or white papers, but as Java annotations that can be checked automatically. The users of a library can determine whether they are violating the library's invariants at runtime, if not even statically at compile time. Considering that writing and using extensible, multi-threaded libraries is one of the most challenging programming tasks, these annotations add considerable value to a library.

The verification of a program's conformity with the threading discipline described by the annotations is performed by automatically inserted bytecode.

To express the invariants, I have elected to use Java annotations, a facility to store metadata that was introduced with Java 5.0 and extended with subtyping in the Extended Annotation Enabled javac (xajavac) [32].

Compared to the alternative approach of using comments with special formatting, annotations have the advantage of being part of the Java language; therefore, their syntax is checked by the Java compiler, and their content is accessible by using the Java API or by reading the well-defined format of a Java class file. Furthermore, it is possible to restrict the places where annotations can be applied: The annotations designed for the invariant checker can only be applied to types (classes and interfaces), methods and constructors. An annotation facility based on comments would involve much more processing to parse annotations and detect badly placed ones.

The invariants introduced in this section are checked at the beginning of a method, right when it is entered, except in constructors, which make a `super` call first. Synchronized methods are changed to regular methods with a `synchronized` block around the entire method body. This allows the invariant checks to be ex-

```
class C {                                                    1
  synchronized void nonS() { /* some code */ }               2
  static synchronized void s() { /* some code */ }           3
}                                                            4
```

Listing 5.1: Synchronized Methods Before Transformation

```
class C {                                                    1
  void nonS() { synchronized(this) { /* some code */ } }     2
  static void s() { synchronized(C.class) { /* some code */ } }  3
}                                                            4
```

Listing 5.2: Methods With Synchronized Blocks After Transformation

ecuted before the lock is acquired: The `synchronized` methods in listing 5.1 are transformed into the methods shown in listing 5.2.

5.1 Annotations and Inheritance

A method can acquire an invariant in three different ways:

1. The method itself is annotated. Listing 5.3 shows and example of this.

2. The same method in a superclass or one of the implemented interfaces has been annotated. Once a method has been assigned an invariant, all overriding implementations will be assigned the same invariant. An example can be found in listing 5.4.

3. The class or interface in which the method is first introduced is annotated. The invariant will be assigned to the same method in all classes or interfaces that extend or implement the annotated class. Examples are shown in listing 5.5. Note that a class annotation is only shorthand for annotating all methods in the class.

```
class C {
 @OnlyEventThread void someMethod() {
    // Method may only be run in the event thread
} }
```

Listing 5.3: Annotated Method

```
class C {
  @OnlyEventThread public void someMethod() {
    // Method may only be run in the event thread
  }
}
interface I {
  // Method may only be run in the event thread
  @OnlyEventThread public void otherMethod();
}
class D extends C implements I {
  public void someMethod() {
    // Method may only be run in the event thread
    // D.someMethod() hasn't been annotated, but C.someMethod(),
    // the same method in the superclass, has
  }
  public void otherMethod() {
    // Method may only be run in the event thread
    // D.otherMethod() hasn't been annotated, but
    // I.otherMethod(), the same method in an implemented
    // interface, has
  }
}
```

Listing 5.4: Method Annotated in Superclass

More formally, let A be an annotation, let m be a method, and let C, D, S be classes or interfaces. Let $m(C)$ be the set of methods defined (i.e. introduced or overridden) in class C. Let $a(C)$ be the set of annotations that are directly attached to class C, i.e. that appear in front of C's class definition, and let $a(C, m)$ be the set of annotations that are directly attached to method m in class C, i.e. that appear in front of m's method definition in the class definition of C. Then $annotations(C, m)$, shown in figure 5.1, is the set of annotations that are applied

```
class B { public void nothing() { /* no annotation at all */ } }
@OnlyEventThread class C extends B {
  public void nothing() { // no annotation at all
    // even though class C is annotated, because nothing()
    // already defined in the superclass
  }
  public void someMethod() { // May only run in event thread
    // C.someMethod() hasn't been annotated, but the class
    // in which it was introduced has
} }
@OnlyEventThread interface I { // May only run in event thread
  // I.otherMethod() has not been annotated, but the interface
  // in which it was introduced has
  public void otherMethod();
}
class D extends C implements I {
  public void nothing() { /* still no annotation at all */ }
  // someMethod() and otherMethod() may only run in event thread
  // they weren't annotated itself, but the superclass or
  // interface where they were already defined, were annotated
  public void someMethod() { /* ... */ }
  public void otherMethod() { /* ... */ }
}
```

Listing 5.5: Annotated Classes

to a method m in class C, either because the method was directly annotated or because the annotations were somehow inherited.

$$\begin{aligned} annotations(C, m) \quad = \quad & \{A : \ \exists S \text{ such that } C <: S, A \in a(S, m)\} \cup \\ & \{A : \ \exists D \text{ such that } C <: D \wedge \\ & \quad \nexists S \text{ such that } D <: S, D \neq S, m \in m(S), \\ & \quad A \in a(D)\} \end{aligned}$$

Figure 5.1 : Set of Annotations

The first subset contains all annotations attached directly to methods. Because of the reflexive property of subtyping $<:$, this subset contains both annotations from methods in superclasses and in class C itself. The second subset contains annotations that are attached to methods because the class in which they were intro-

duced was annotated. More precisely, the second subset consists of all the annotations that meet the following three criteria: the method exists in the class, superclass, or an implemented interface; that class, superclass, or interface is annotated with the annotation; and the method has not already been introduced in a class or interface higher up.

This structure of inheritance of invariants is crucial in enforcing invariants in subclasses. It allows library developers to design APIs that users can extend while still ensuring that the original threading discipline is maintained.

The decision to let annotations on classes only affect methods that are first introduced in that class or one of its subclasses helps localize the effect of an annotation. The users of a library are free to introduce their own invariants, in addition to the library's invariants, without accidentally strengthening the invariants of methods that have already been defined in a library superclass or interface.

5.2 Predicate Annotations without Subtyping

After initially implementing a limited set of hard-coded annotations, I noticed that only a small set of the desirable invariants could be expressed, particularly when annotating the model object associated with a GUI component: For instance, it is easy to specify that a `JTable` should only be accessed by the event thread once the component has been realized, but how would a developer do the same for the `TableModel` object that contains the data displayed in the table? The model object does not have a reference to the GUI component. Predicate annotations remedy this problem.

A predicate annotation is an annotation that itself is annotated by one of the two meta-annotations `@PredicateLink` and `@Combine`. These meta-annotations mark an annotation as predicate annotation. In the absence of subtyping for annotations, the meta-annotations also specify the way the predicate annotation behaves.

Annotations marked as `@PredicateLink` may only contain primitive data members, strings, class objects, enumerations, and arrays of the types just mentioned; they may not contain other annotations or arrays of annotations as members. Annotations marked with `@Combine`, on the other hand may only contain other annotations or arrays of annotations (in fact, the annotations must even be predicate annotations, i.e. they must have either a `@PredicateLink` or a `@Combine` meta-annotation); annotations marked with `@Combine` may not contain primitive data members, strings, class objects, enumerations, or arrays of those types.

5.2.1 Predicate Link Annotations

Listing 5.6 shows the definition of the `@PredicateLink` meta-annotation. The `@PredicateLink` meta-annotation establishes a link between the annotation it is marking and a static method returning a `boolean` (the predicate method, specified by a `Class` instance and a method name). This method is called, and if it returns `false`, a violation has occurred. The predicate method must be in a completely static context, which means the method cannot be in an anonymous inner class or a non-static inner class; methods in static inner classes, however, can be used. To be precise, it has to be possible to call the predicate method from the beginning of every method that is affected by the annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface PredicateLink {
  /** Class containing the predicate method.
    * @return class with the predicate method. */
  Class value();
  /** Name of the predicate method. If not
    * specified, the name "check" is assumed.
    * @return name of the predicate method. */
  String method() default "check";
  /** Whether method arguments should be passed
    * to the predicate.
    * @return true if arguments should be passed. */
  boolean arguments() default false;
}
```

Listing 5.6: `@PredicateLink` Meta-Annotation

The method is specified using the `value()` and `method()` members of the `@PredicateLink` annotation. The former specifies the class that contains the predicate method. The latter is optional and specifies the method name; if no method name is specified, `"check"` is assumed. The third member, `arguments()`, decides whether the method arguments should be passed.

Listing 5.7 shows the annotation definition, predicate method, and usage site of the `@ThreadWithNameNotPassed` annotation. The annotation specifies that it is not permissable to pass a `Thread` object to the method whose `Thread.getName()` method returns a certain string, for instance `"Disallowed"`.

The example in listing 5.7 shows that the predicate method can receive data from the annotation, in this case the name of a thread in the `value()` member, and the arguments from the method, passed in the `Object[] args` parameter. The requirements on the predicate methods and the bytecode rewriting procedure were described in detail in my Master's thesis [34].

```
@PredicateLink(value=Predicates.class,                                  1
               method="checkThreadWithNameNotPassed",                   2
               arguments=true)                                          3
public @interface ThreadWithNameNotPassed {                             4
  String value(); // name of thread not to pass                         5
}                                                                       6
class Predicates {                                                      7
  public static boolean checkThreadWithNameNotPassed                    8
      (Object this0, String value, Object[] args) {                     9
    // this0 is bar's this                                             10
    // value is the name of the thread not to pass                     11
    // args contains the arguments to bar                              12
    for(Object arg: args) {                                            13
      if (arg instanceof Thread) {                                     14
        Thread t = (Thread)args;                                       15
        if (t.getName().equals(value)) return false;                   16
      }                                                                17
    }                                                                  18
    return true;                                                       19
  }                                                                    20
}                                                                      21
class Foo {                                                            22
  @ThreadWithNameNotPassed("Disallowed"                                23
  public void bar(String dummy0, Thread t, Object dummy1) {            24
    // ...                                                             25
  }                                                                    26
}                                                                      27
```

Listing 5.7: Predicate Annotation with a Member, Arguments Passed

```
@PredicateLink(value=Predicates.class,                                  1
               method="checkThreadWithNameNotPassed",                   2
               arguments=true)                                          3
public @interface ThreadWithNameNotPassed {                             4
  String value(); // name of thread not to pass                         5
}                                                                       6
```

Listing 5.8: Annotation Definition of Listing 5.7

```
class Predicates {                                                      1
  public static boolean checkThreadWithNameNotPassed                    2
      (Object this0, String value, Object[] args) {                     3
    // this0 is bar's this                                              4
    // value is the name of the thread not to pass                      5
    // args contains the arguments to bar                               6
    for(Object arg: args) {                                             7
      if (arg instanceof Thread) {                                      8
        Thread t = (Thread)args;                                        9
        if (t.getName().equals(value)) return false;                    10
      }                                                                 11
    }                                                                   12
    return true;                                                        13
  }                                                                     14
}                                                                       15
```

Listing 5.9: Predicate Method of Listing 5.7

Predicate annotations consist of three individual parts: The annotation definition, the predicate method, and the actual usage site of the annotation. While listing 5.7 shows all three parts together, the reader should keep in mind that the annotation definition and the predicate method are usually written by the library developer, and the application developer only needs to work with the annotation usage.

To show the simplicity of this scheme, the individual parts from listing 5.7 have been separated and are shown in listings 5.8, 5.9 and 5.10.

```
class Foo {
  @ThreadWithNameNotPassed("DisallowedThread"
  public void bar(String dummy0, Thread t, Object dummy1) {
    // ...
  }
}
```

Listing 5.10: Usage Site of Listing 5.7

```
@And({
  @Or({
    @NotThreadWithName("foo"),
    @NotThreadWithName("bar")}),
  @Not(@NotThreadWithGroupName("main")),
  @NotThreadWithID(5)})
void someMethod() { /* ... */ }
```

Listing 5.11: Ideal, But Unachievable Usage Example

5.2.2 Combine Annotations

`@Combine` meta-annotations were created because users longed for the ability to perform Boolean operations on predicate annotations to create larger, more complex compound annotations. The example shown in 5.11 would have been the ideal usage, but as discussed in section A.2 in the appendix, this cannot be achieved using Java's annotation system. Without subtyping for annotations, the `@Combine`-style annotations were the best solution I could find.

The definition of the `@Combine` meta-annotation can be found in listing 5.12. The `@Combine` meta-annotation has two members, `value()` and `arguments()`. The former decides if the member annotations of the annotation marked by the `@Combine` meta-annotation should be merged using "and", "or", "xor", "not", or "implies". The second member, `@Combine.arguments()`, determines whether

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.ANNOTATION_TYPE})
public @interface Combine {
  public static enum Mode {
    AND, OR, XOR, NOT, IMPLIES;
  }
  /** Operation used to combine the member predicate
    * annotations, i.e. AND, OR, XOR, NOT, IMPLIES.
    * @return combining operation */
  Mode value() default Mode.OR;
  /** Whether method arguments should be passed
    * to the predicate.
    * @return true if arguments should be passed. */
  boolean arguments() default false;
}
```

Listing 5.12: `@Combine` Meta-Annotation

the method arguments should be passed to the predicate methods, just like the `arguments()` member of the `@PredicateLink` meta-annotation did.

Because `@Combine`-style annotations are used to create compounds of other predicate annotations, the only members allowed in a `@Combine`-style annotation are other predicate annotations or arrays of predicate annotations; primitive data, strings, class objects, enumerations, or arrays of the aforementioned types are not allowed. The member annotations or the elements in the member array must also have been annotated with one of the meta-annotations `@PredicateLink` or `@Combine`.

At the usage site, there is no difference between using a `@PredicateLink`-style or a `@Combine`-style annotation. A `@Combine`-style annotation, however, does not have a designated predicate method, like the one specified by a `@PredicateLink` meta-annotation. When the invariant checker encounters a usage site of a `@Combine`-style annotation, it automatically generates a predicate method that combines the member annotations using the Boolean operation

```
@Combine(value=Combine.Mode.OR)                                    1
public @interface NotThreadWithNameInGroup {                       2
  NotThreadWithName name();                                        3
  NotThreadWithGroupName group();                                  4
}                                                                  5
@NotThreadWithNameInGroup(                                         6
  name=@NotThreadWithName("fee"),                                  7
  group=@NotThreadWithGroupName("bar"))                            8
void someMethod() { /* ... */ }                                    9
```

Listing 5.13: A `@Combine` Annotation That Combines Member Annotations using "or"

specified by `@Combine.value()`. Listing 5.13 shows and example `@Combine`-style annotation that combines the predicate annotations `@NotThreadWithName` and `@NotThreadWithGroupName` using a Boolean "or".

More details about `@Combine`-style annotations were described in my Master's thesis [34].

5.3 Predicate Annotations with Subtyping

After completing my Master's thesis, I examined the javac compiler to look for a way to improve Java annotations.

The annotations are implemented as interfaces, encoded as regular Java class files. Annotation interfaces implicitly implement the `java.lang.annotation.Annotation` interface, but otherwise do not support subtyping. Even though Java interfaces can extend multiple other interfaces, as shown below, the `extends` clause is not permitted for annotations.

Furthermore, annotations may only contain primitive data, strings, class literals (Example: `MyClass.class`), enumerations, other annotations, and arrays of the beforementioned as members. Specifically, most objects cannot be stored in

an annotation (except for instances of `String`, `Class<?>` and enumeration types). `java.lang.annotation.Annotation` is the implicit supertype of all annotations, but not considered an annotation itself; therefore, it cannot be a member of an annotation.

The lack of subtyping leads to poor code reuse: It is impossible to write one annotation that accepts more than one kind of annotation for a certain member; the exact type of all members has to be determined in the annotation. Even though the `@Or` and `@And` annotations in listing 5.11 work the same for all kinds of member annotations, I had to define them anew for every new class of invariant annotation. Mixing different invariant annotations was even more complicated and resulted in the `@Combine` meta-annotation. This is the same problem that abstract data types such as lists presented and that was so elegantly solved, either with subtyping or generics: The same list class can be used for all types of data.

When I examined the Java compiler and the class files it produces for annotations, I found nothing that prevented subtyping for annotations. In the compiler, I merely had to remove a few checks to allow the `extends` clause; the class file format could remain completely unchanged.

xajavac allows the `extends` clause for annotations. Just like with interfaces, one or more annotations may be extended. The types that are extended, however, all have to be annotations themselves. Extending classes or interfaces that are not annotations results in an error. If no `extends` clause exists, then `java.lang.annotation.Annotation` is implicitly made the superclass, just like in standard Java. Since `java.lang.annotation.Annotation` is not an annotation itself, it cannot be mentioned in the `extends` clause of an annotation.

Allowing subtyping for annotations greatly simplified both the implementation of the invariant checker and the use of the invariants. Listing 5.14 shows that `@Combine`-style meta-annotations are not necessary anymore, and that Boolean operators like `@And` and `@Or` are `@PredicateLink`-style annotations just like any other invariant annotation. The size of the implementation was reduced by a factor of 3, measured in terms of lines of code, and 86 pre-defined annotations could be replaced by just 19, mostly because Boolean operators did not have to be defined for each invariant separately.

5.4 Results

To evaluate the efficacy and ease of use of the invariant checker and the annotations, I added invariant annotations to two different versions of DrJava [31], a version from March 26, 2004, and a version from September 2, 2006. These versions were chosen because they marked two stable releases of DrJava. Since the DrJava development team had recently made many changes to improve concurrent behavior of the application, I expected that the older version would have more invariant violations than the more recent version.

The process of annotating existing code was primarily guided by source code comments and Javadoc comments present in the source code. It should be noted that I probably missed some opportunities for annotations and only expressed a subset of the invariants actually present in the program. I also faced some problems building and running the 2004 version of DrJava, which was written before the final version of Java 5.0 was released; as a result, some unit tests could not be run.

```
public @interface InvariantAnnotation { }

@PredicateLink(value=Predicates.class, method="checkAnd",
               arguments=true)
public @interface And extends InvariantAnnotation {
  public abstract InvariantAnnotation[] value();
}

@PredicateLink(value=Predicates.class, method="checkOr",
               arguments=true)
public @interface Or extends InvariantAnnotation {
  InvariantAnnotation[] value();
}

public class Predicates {
  public static boolean checkOr(Object thisObject,
                                Object[] methodArgs,
                                InvariantAnnotation[] value) {
    for(InvariantAnnotation ia: value) {
      if (checkInvariantAnnotation(ia, thisObject, methodArgs)) {
        return true;
      }
    }
    return false;
  }

  public static boolean checkAnd(Object thisObject,
                                 Object[] methodArgs,
                                 InvariantAnnotation[] value) {
    for(InvariantAnnotation ia: value) {
      if (!checkInvariantAnnotation(ia, thisObject,
                                    methodArgs)) {
        return false;
      }
    }
    return true;
  }
}
```

Listing 5.14: Boolean Operations on Invariants Using Annotations with Subtyping

Table 5.1 shows the number of unit tests for the two versions that passed successfully, failed, or could not be run because of problems with Java 5.0, as well as the total number of unit tests. It is evident that many tests were added to the unit testing suite of DrJava between the two versions.

	3/26/2004 Version	9/2/2006 Version
Unit Tests Passed	610	881
Unit Test Failures	36	0
Could Not Run	90	0
Total Unit Tests	736	881

Table 5.1 : Unit Tests

Table 5.2 shows the total number of invariant checks, the number of passed and failed checks, and the percentage of failed checks during the execution of the entire test suite. While there were more check failures in the 2006 version in absolute terms, the percentage of failed invariant checks was significantly lower in the newer version, reflecting the development team's perception that concurrent behavior had improved. The number of invariant checks was lower in the 2004 version because there were fewer comments that dealt with concurrency, which made the annotation process for the 2004 version more difficult.

This lack of documentation is corroborated by the information in table 5.3: On the one hand, the source base and the number of unit tests grew substantially, and on the other hand, the term "event thread", a concept central to Java AWT and Swing threading disciplines, was mentioned hardly at all in the 2004 version, but frequently in the 2006 version.

	3/26/2004 Version	9/2/2006 Version
Invariant Checks Failed	965	3796
Invariant Checks Passed	4161	30616
Total Invariant Checks	5116	34412
Percentage Failed	18.83	11.03

Table 5.2 : Invariant Checks and Violations

	3/26/2004 Version	9/2/2006 Version
KLOC	107	129
Total Unit Tests	736	881
Mentions "event thread"	1	99

Table 5.3 : Other Information

It should also be noted that the 34,412 invariant checks passed in the 2006 version did not prolong the testing process measurably. In general, as long as there was sufficient information available to establish concurrency invariants, it was easy to add annotations and have them checked. It was also simple to annotate only a part of the codebase without annotating the whole codebase, and it was easy to carry out the invariant checking in addition to running the unit testing suite. I expect that it will be much easier to annotate a program at the same time it is written, when the programmers are actually the most aware of the required invariants.

Chapter 6

Bytecode Rewriting

All of the pieces of the framework use bytecode rewriting to some degree: Java source files are compiled with the regular Java compiler, if possible with debug information, and then analyzed and rewritten.

The alternative would have been to rewrite Java source code, but this approach has several disadvantages:

- It is more difficult to parse a Java source file due to all the variations that are allowed.
- Some of the instrumentations that can be performed do not have a corresponding Java source equivalent. For example, it is impossible in Java to just emit a `MONITORENTER` instruction without a matching `MONITOREXIT` instruction.
- Sometimes, the Java source simply is not available or it is not advisable to recompile it, as is the case with many of the classes of the Java API.
- Performing the instrumentation on-the-fly, using a custom class loader, would have been much more difficult if the changes were made to Java source and not to class files. The compiler would have to be invoked at runtime from within a class loader, which is possible but expensive, as the multi-stage Java programming language Mint demonstrates [56].

Another option would have been to modify the Java compiler and runtime environment, but this was ruled out early on so the project could target as many platforms as possible and therefore maximize the potential user base.

The kind of analysis and rewriting that is performed depends on the required task. In general, class files are rewritten one at a time; the actual process has been abstracted out using several object-oriented design patterns, namely strategy, decorator and composite [13], so that each instrumentation can implement the necessary changes as it sees fit. Very often, the instrumentation strategy cycles through all the methods in a class and changes them.

`IInstrumentationStrategy` is the base interface, and all instrumentation strategies need to implement its two methods shown in listing 6.1. `instrument` will be called once per class file; `done` is called only once, at the end of the instrumentation, when all classes have been processed.

The `isReady` method is called to determine if a class file is ready to be instrumented. If the class file is not ready, it is placed at the end of the queue of files to process, which allows the instrumentation strategy to change the order classes are instrumented. The execution logging strategy in chapter 4, for example, examines all classes that need to be logged and then constructs a table in `ExecutionLog` class; the `ExecutionLog` class therefure must be processed last. To implement this, the `isReady` method returns `false` for the `ExecutionLog` class until all other classes have been instrumented.

Cycles in the order of dependencies expressed by the `isReady` method are not allowed, and the framework throws an exception if one complete iteration through the queue of class files does not result in a change in the queue contents.

There are several helpful classes and interfaces that allow better code reuse:

```
public interface IInstrumentationStrategy {
  public void instrument(ClassFile cf);
  public boolean isReady(ClassFile cf);
  public void done();
}
```

Listing 6.1: `IInstrumentationStrategy` Source

```
public abstract class ConditionalStrategy
  implements IInstrumentationStrategy {
  IInstrumentationStrategy _decoree;
  public ConditionalStrategy(IInstrumentationStrategy decoree) {
    _decoree = decoree;
  }
  public void instrument(ClassFile cf) {
    // ...
    if (apply(cf)) {
      // ..
      _decoree.instrument(cf);
    }
  }
  public boolean isReady(ClassFile cf) { return
    _decoree.isReady(cf);
  }
  public void done() { _decoree.done(); }
  public abstract boolean apply(ClassFile cf);
}
```

Listing 6.2: `ConditionalStrategy` Source

`CompoundStrategy` bundles several `IInstrumentationStrategy` instances and runs them one after the other. A `ConditionalStrategy`, shown in listing 6.2, also contains another `IInstrumentationStrategy`, but it will only execute the strategy if the `ConditionalStrategy.apply` method returns `true`.

Using a `ConditionalStrategy`, a developer can apply an instrumentation only to class files that, for example, reside in the `java.lang` package. There is also an additional interface that instrumentation strategies can implement, `IScannerStrategy`, which adds another method to be implemented and whose

```
public interface IScannerStrategy extends                          1
    IInstrumentationStrategy {
  public interface IScanResult {                                    2
      public String getPropertyName();                             3
  }                                                                 4
  public List<? extends IScanResult> getScanResults();             5
}                                                                   6
```

Listing 6.3: `IScannerStrategy` Interface

purpose it is to return data gathered during the instrumentation. The interface's definition is shown in listing 6.3.

The classes implementing `IScannerStratey` typically cache data from the instrumentation of one class to the next, and possibly process it when the `done` method is called. Another common use of the `IScannerStrategy` is storing minor errors that should be relayed to the user but that should not entirely terminate the instrumentation, as an exception would.

Most instrumentation strategies also take a `List<String>` as parameter in their constructors: This list allows the user to pass values to the instrumentation strategies and, for example, determine whether backup files should be created before a class file is changed.

To parse, analyze and modify Java class files, I decided to use my own library that I created as part of the work for my Master's thesis [34], even though alternatives such as BCEL [2] and ASM [28] existed. It took some time to write the library, but I believe this effort was necessary to get acquainted with all parts of the Java class file format and the intricacies of the JVM.

The framework consists of a mixture of high-level classes that employ object-oriented design patterns like strategies, compounds and visitors, as well as low-level

constructs for the compact representation of instructions, methods and classes. All features of Java 5.0 and 6.0 are fully supported by the library.

6.1 Offline and On-the-Fly Instrumentation

Regardless of which instrumentation strategy is used, instrumentation can be done either offline, i.e. after compile time but before runtime; or at runtime using a custom class loader that rewrites the classes just as they are needed.

To instrument classes offline, a set of class files, directories, and jar files is passed as argument to the `FileInstrumentor` program, together with the name of the `IInstrumentationStrategy` that should be applied. The `FileInstrumentor` then processes all class files that were specified and replaces the originals with instrumented copies. After the instrumentation, the classes can be used just as if nothing had changed.

Offline instrumentation is faster, safer, more accurate and more general than on-the-fly instrumentation: It is faster because caching changes is easier, and because the JVM does nothing else besides instrumenting class files.

Performing the instrumentation offline is safer and more accurate than changing the class files on-the-fly because the instrumentation cannot have any side effects on computations that happen concurrently. With on-the-fly instrumentation, care must be taken to minimize the impact the custom class loader has on the rest of the program. If the custom class loader affected the behavior of the program being instrumented and changed the outcome of a particular unit test, that would negate the effort of the framework.

Finally, several classes in the Java API are considered "protected" and cannot be changed on-the-fly. In order to change them, the instrumentation has to be performed offline. For all of these reasons, it is recommended to instrument the Java API (usually called `rt.jar` on Windows and Linux, or `classes.jar` and `ui.jar` on Mac OS X) offline and create instrumented copies to be used instead of the original API files. To use the instrumented copies, they have to be placed at the beginning of Java's boot classpath using the `-Xbootclasspath/p` option. The framework provides GUI tools to assist the user with this.

6.2 Local and Global Instrumentation

Each instrumentation strategy, each change of the program, can be classified as either local or global, depending on what parts of the program need to be modified to achieve the desired effect.

The changes from chapter 5 to check whether the program has violated the threading discipline are an example of local instrumentation. Bytecode is inserted in one place, at the beginning of a method, but the results are observable throughout the entire program, at every call site of the method. Converting `synchronized` methods to methods with a `synchronized` block is another example of local instrumentation. A change is made in one method only, the rest of the program does not have to be modified.

Not all changes can be made in this way, though. In order to instrument all synchronization points of a program, for example, it is necessary to know when a program calls the `Object.wait`, `Object.notify`, and `Object.notifyAll` methods. The easiest way to achieve this would be to insert the bytecode processing the

synchronization point into these methods; in the current Java API, however, these methods are native and therefore do not contain bytecode.

The next easiest way to be notified of every call to the methods above would be to rename the original methods, e.g. rename the `wait` method in the `Object` class to `waitOriginal`, and then put a method with the original name in its place, a method that processes the invocation appropriately and then forwards the call to the renamed method. That way, the changes would still be localized to the `Object` class alone; all other classes could remain unchanged and would nonetheless call the method we put in place.

This renaming approach works for some methods, but it does not help in the case of the `wait`, `notify`, and `notifyAll` methods in the `Object` class. These methods are native, and linking the native code to the methods requires the methods to always have the original name.

To still be notified of calls to these methods, an instrumentation strategy has to create forwarding methods, for example `Object.waitForward`, that do the processing and then call the native methods. Unfortunately, all the other classes still call the original method; therefore, all call sites in all classes need to be changed to invoke the added forwarding method instead. That makes this kind of instrumentation global; the changes are not localized to a single class anymore, but affect every class that uses the changed method. Local instrumentation is preferable to global instrumentation since it reduces the number of times code has to be rewritten.

In chapter 2, ConcJUnit used bytecode rewriting to detect unjoined threads; the scheduling component of Concutest in chapter 3 inserted bytecode for random delays, which allows tests to be executed under varying schedules. In the last two chapters, bytecode rewriting was used to log the execution of methods and to verify that invariants were being maintained. Bytecode rewriting proved to be valuable in all parts of the testing framework.

Chapter 7

Conclusion

This thesis described a new framework called Concutest that can effectively apply unit testing to concurrent programs, which are difficult to develop and debug. Test-driven development, a practice that in the past has only been productive for programs with a single thread of control, can now be used with concurrent programs.

The Concutest framework

- improves JUnit to recognize errors in all threads, a necessary development without which all other improvements are futile,
- places some restrictions on the programs to facilitate automatic testing,
- provides an invariant checker that reduces programmer mistakes by checking thread disciplines,
- simplifies writing unit tests that need to log the execution of portions of code that can only be observed using side effects, which is frequently the case in concurrent programs, and
- re-runs the unit tests with randomized schedules to simulate the execution under different conditions and on different machines, increasing the probability that errors are detected.

7.1 Future Work

The Concutest framework is an effective tool for programmers; however, it also represents an interesting platform for future research.

For example, choosing the enabled subset of instrumentation sites for delays is still a haphazard activity. It would be worthwhile to study the interactions between different synchronization points more closely.

Performing an effective, light-weight static analysis to determine which fields and arrays could actually be shared across threads may minimize the time spent in unnecessary delays.

Determining field sharing could also be done dynamically: Instead of inserting delays, the framework could keep track of the thread that accesses a field, and only start using delays when it becomes clear that the field is shared. Using an atomic `compareAndSet` operation, I expect this to take less time in the unshared case than an unnecessary delay.

Varying the delay durations and the probability that one is inserted at all could be insightful. For example, it could be interesting to start with shorter, less frequent delays, and increase duration and frequency every time a test passes without errors. There also seems to be an interaction between the number of processor cores and the lengths of the delays necessary to effectively detect errors, and it may not be necessary to use long delays for a processor with many cores.

In spite of these research questions, the Concutest framework developed as part of this thesis provides useful tools for concurrent programs, enabling developers to test programs more reliably.

The tools described in this thesis work on all three major platforms: Windows, Linux, and Mac OS X.

All source code is open source and available at: `http://www.concutest.org/` [33].

Bibliography

[1] Apache. log4j Logging Services. `http://logging.apache.org/log4j/1.2/`.

[2] Apache Jakarta Project. BCEL Byte Code Engineering Library Website. `http://jakarta.apache.org/bcel/`.

[3] ArgoUML Project. ArgoUML Website. `http://argouml.tigris.org`.

[4] Binder, W., Hulaas, J., and Moret, P. Advanced java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java* (New York, NY, USA, 2007), PPPJ '07, ACM, pp. 135–144.

[5] Bowers, A. N., Sangwan, R. S., and Neill, C. J. Adoption of xp practices in the industrya survey: Research sections. *Softw. Process 12* (May 2007), 283–294.

[6] Bruening, D. L. *Systematic Testing for Multithreaded Programs.* Master's thesis, MIT, Cambridge, MA, USA, 1999.

[7] CalFuzzer Project. CalFuzzer Website. `http://srl.cs.berkeley.edu/~ksen/calfuzzer/`.

[8] Click, C. Highly Scalable Java Website. `http://sourceforge.net/projects/high-scale-lib/`.

[9] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S. Multithreaded java program test generation. *IBM Syst. J. 41* (January 2002), 111–125.

[10] Findler, R. B., Latendresse, M., and Felleisen, M. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2001), ESEC/FSE-9, ACM, pp. 229–236.

[11] Fischer, R. jconch Website. `http://code.google.com/p/jconch`.

[12] Flanagan, C., and Freund, S. N. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 121–133.

[13] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, New York, NY, USA, 1994.

[14] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12* (October 1969), 576–580.

[15] Jefferies, R. Xprogramming.com. `http://www.xprogramming.com`.

[16] JFreeChart Project. JFreeChart Website. `http://www.jfree.org/jfreechart/`.

[17] JOSHI, P., NAIK, M., PARK, C.-S., AND SEN, K. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2009), CAV '09, Springer-Verlag, pp. 675–681.

[18] JUNIT PROJECT. JUnit Website. `http://www.junit.org`.

[19] KARAORMAN, M., AND ABERCROMBIE, P. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Form. Methods Syst. Des. 27* (November 2005), 275–312.

[20] KARAORMAN, M., HOLZLE, U., AND BRUNO, J. jcontractor: A reflective java library to support design by contract. Tech. rep., Santa Barbara, CA, USA, 1999.

[21] KAWAGUCHI, K. Parallel JUnit. `https://parallel-junit.dev.java.net/`.

[22] KERFOOT, E., AND MCKEEVER, S. Checking concurrent contracts with aspects. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, ACM, pp. 2523–2530.

[23] KODERS, INC. Koders.com. `http://www.koders.com`.

[24] LI, L., AND VERBRUGGE, C. A practical mhp information analysis for concurrent java programs. In *of Lecture Notes in Computer Science* (2004), Springer, pp. 194–208.

[25] LISKOV, B. H., AND WING, J. M. *Behavioural subtyping using invariants and constraints.* Cambridge University Press, New York, NY, USA, 2001, pp. 254–280.

[26] MITCHELL, R., MCKIM, J., AND MEYER, B. *Design by contract, by example.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[27] NAIK, M., AIKEN, A., AND WHALEY, J. Effective static race detection for java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2006), PLDI '06, ACM, pp. 308–319.

[28] OBJECT WEB. ASM Java Bytecode Manipulation Framework. `http://asm.objectweb.org/`.

[29] ORACLE. Hudson: a continuous integration server. `http://wiki.hudson-ci.org`.

[30] ORACLE. Java Logging Technology Developer Guide. `http://download.oracle.com/javase/6/docs/technotes/guides/logging/index.html`.

[31] RICE JAVAPLT. DrJava Web Site. `http://drjava.org`.

[32] RICE JAVAPLT. xajavac Extended Annotation Enabled javac. `http://www.cs.rice.edu/~mgricken/research/xajavac/`.

[33] RICKEN, M. Concutest Project Website. `http://www.concutest.org`.

[34] RICKEN, M. *A Framework for Testing Concurrent Programs.* Master's thesis, Rice University, Houston, TX, USA, 2007.

[35] RICKEN, M., AND CARTWRIGHT, R. Concjunit: unit testing for concurrent programs. In *Proceedings of the 7th International Conference on Principles*

and Practice of Programming in Java (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 129–132.

[36] RUMPE, B., AND SCHRÖDER, A. Quantitative survey on extreme programming projects. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002* (2002), pp. 26–30.

[37] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*, vol. 15. in ACM Trans. Comput. Syst., **15**, ACM Press, New York, NY, USA, 1997.

[38] SEN, K. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), PLDI '08, ACM, pp. 11–21.

[39] SOOT PROJECT. Soot Web Site. `http://www.sable.mcgill.ca/soot/`.

[40] SPACCO, J., AND PUGH, W. Helping students appreciate test-driven development (tdd). In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), OOPSLA '06, ACM, pp. 907–913.

[41] STASKO, J. Animating algorithms with xtango. *SIGACT News 23* (May 1992), 67–71.

[42] STOLLER, S. D. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)* (July 2002), vol. 70(4) of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[43] Sun Microsystems, Inc. Java Language Specification. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[44] Sun Microsystems, Inc. Java Virtual Machine Specification. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html.

[45] Sun Microsystems, Inc. Javadoc for Thread.UncaughtExceptionHandler. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.UncaughtExceptionHandler.html.

[46] Sun Microsystems, Inc. JSR 133: Java Memory Model and Thread Specification Revision. http://jcp.org/en/jsr/detail?id=133.

[47] Sun Microsystems, Inc. JSR 308: Annotations on Java Types. http://www.jcp.org/en/jsr/detail?id=308.

[48] Sun Microsystems, Inc. SUN JDC Tech Tips, March 28, 2000: Why Use Threads? Protecting Shared Resources with Synchronized Blocks. Minimizing the Overhead of Synchronized Blocks. Retrieved from http://javaservice.net/~java/bbs/read.cgi?m=devtip&b=jdc&c=r_p_p&n=954297433 – originally published at http://developer.java.sun.com/developer/TechTips/2000/tt0328.html.

[49] Szeder, G. Unit testing for multi-threaded java programs. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging* (New York, NY, USA, 2009), PADTAD '09, ACM, pp. 4:1–4:8.

[50] TestNG Project. TestNG Website. http://testng.org.

[51] The Eclipse Foundation. AspectJ. http://www.eclipse.org/aspectj/.

[52] Villazón, A., Binder, W., and Moret, P. Aspect weaving in standard java class libraries. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java* (New York, NY, USA, 2008), PPPJ '08, ACM, pp. 159–167.

[53] Villazón, A., Binder, W., Moret, P., and Ansaloni, D. Major: rapid tool development with aspect-oriented programming. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java* (New York, NY, USA, 2009), PPPJ '09, ACM, pp. 125–128.

[54] Visser, W., Havelund, K., Brat, G., and Park, S. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering* (Washington, DC, USA, 2000), ASE '00, IEEE Computer Society, pp. 3–.

[55] Walker, D., Zdancewic, S., and Ligatti, J. A theory of aspects. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2003), ICFP '03, ACM, pp. 127–139.

[56] Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., and Taha, W. Mint: Java multi-stage programming using weak separability. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2010), PLDI '10, ACM, pp. 400–411.

Appendix A

Suggestions for Improving Java Annotations

During my extensive work with Java annotations, I have discovered several shortcomings in the way the annotations are implemented in the Java compiler, the Java runtime, and the Java Language Specification [43]:

- It is illegal to use the same annotation more than once per target.
- There is no subtyping for annotations.

These shortcomings are described in greater detail in the sections below. As previously predicted [34], these issues were easy to address: The Extended Annotation Enabled javac compiler (xajavac) [32] supports both repeated annotations and annotations with subtyping.

The Java Specification Request "Annotations on Types" (JSR 308) [47] cited my work on annotations but failed to remedy these problems, mostly because these problems were considered out of scope for the specification request.

A.1 Repeated Annotations

In Java, it is currently illegal to attach the same annotation more than once to a target. Allowing repeated annotations would make specifying several similar pieces of metadata much easier. Listing A.1 shows the most intuitive way to spec-

```
@LogTheMethod("MyClass.foo")                                    1
@LogTheMethod("MyClass.bar")                                    2
public void testLogged() { /* ... */ }                          3
```

Listing A.1: Repeated Annotations

ify that a test method should log the execution of two methods, `MyClass.foo` and `MyClass.bar`, but the Java compiler will reject the repeated annotation.

The execution logging system provides a work-around for this problem: Listing A.2 shows how the desired logging request from listing A.1 can be achieved using an array of helper annotations. Specifying multiple pieces of metadata that use the same annotation is therefore possible, but not as convenient as it could be.

As xajavac demonstrates, there is no compelling reason against allowing repeated annotations. To allow reflection with more than one annotation per type, the `getAnnotations(Class<T> annotationClass)` method should be added to the existing `getAnnotation(Class<T> annotationClass)` and `getAnnotations()` methods in `java.lang.reflect.AccessibleObject`:

```
public <T extends Annotation> T
  getAnnotation(Class<T> annotationClass);

public <T extends Annotation> T
  getAnnotations(); // all annotations

public <T extends Annotation> T[]
  getAnnotations(Class<T> annotationClass);
```

The semantics of the original `getAnnotation` should be changed to return the first annotation if more than one exists. Allowing repeated annotations could have made `@LogTheMethod` and `@LogTheClass` annotations much less verbose.

```
@LogTheMethod({                                          1
  @MethodDesc("MyClass.foo"),                            2
  @MethodDesc("MyClass.bar")                             3
})                                                       4
public void testLogged() { /* ... */ }                   5
```

Listing A.2: Annotation Array as Alternative

```
@interface Named {                                       1
  String value();                                        2
  boolean regex() default false;                         3
}                                                        4
@interface NotThreadWithName extends Named { }           5
@interface NotThreadWithGroupName extends Named { }      6
```

Listing A.3: Extending Annotations

A.2 Subtyping for Annotations

In Java, an annotation cannot extend another annotation, even though annotations are handled in a very similar way as interfaces. It could often be useful to extend one annotation and add additional elements, as listing A.3 shows.

The `extends` clause is not allowed in annotation declarations, even though allowing it could often increase code reuse. The most unfortunate result of the lack of subtyping for annotations is that two annotations do not have a common base class and therefore cannot be treated abstractly. All annotations implement the interface `java.lang.annotation.Annotation`, but that interface itself is not an annotation and therefore cannot be used as member of an annotation.

If subtyping is allowed for annotations, sections 4.1.4 and 5.3 demonstrate that it is trivial to develop concise annotations that perform the Boolean operations "and", "or", and "not".

To allow reflection for annotations with subtyping, the runtime library should add the `getAnnotations` method as described above, but also provide ways to look for exact class matches and ignore subtypes:

```
// include subtypes
public <T extends Annotation> T[]
  getAnnotations(Class<T> annotationClass);

// option to include/exclude subtypes
public <T extends Annotation> T[]
  getAnnotations(Class<T> annotationClass, boolean includeSubtypes);
```

The semantics of the original `getAnnotation` method should probably be changed to ignore subtypes and to return the first annotation that matches the provided class, or `null` if none is found.

A.3 Extended Annotation Enabled javac (xajavac)

The xajavac Extended Annotation Enabled javac compiler is available under the Java Research License at:

`http://www.cs.rice.edu/~mgricken/research/xajavac`

and

`http://ricken.us/research/xajavac`

To use the modified compiler, download the `xajavac.jar` file (using the xajavac Binaries link below) and invoke it using

```
java -jar xajavac.jar <arguments>
```

where `<arguments>` stands for the arguments usually passed to javac.

xajavac allows the `extends` clause for annotations as well as repeated annotations. Just like with interfaces, one or more annotations may be extended. The

```
AnnotationTypeDeclaration:                                          1
  @ interface Identifier AnnotationTypeBody                         2
```

Listing A.4: Original Java Grammar for Annotations

```
AnnotationTypeDeclaration:                                          1
  [final] @ interface Identifier [extends AnnotationTypeList]       2
     AnnotationTypeBody
AnnotationTypeList:                                                 3
  AnnotationType {   ,   AnnotationType}                            4
AnnotationType:                                                     5
  Identifier                                                        6
```

Listing A.5: Changed Grammar to Allow Subtyping for Annotations

types that are extended, however, all have to be annotations themselves. Extending classes or interfaces that are not annotations results in an error. If no extends clause exists, then `java.lang.annotation.Annotation` is implicitly made the superclass, just like in standard Java. Since `java.lang.annotation.Annotation` is not an annotation itself, it cannot be mentioned in the **`extends`** clause of an annotation.

Formally, I have changed the grammar of Java from what has been published in the Java Language Specification, Third Edition [43] in just one place. Listing A.4 shows the original portion of the grammar, and listing A.5 shows the changes incorporated into xajavac.

There is a context-sensitive requirement for the **`AnotationType`** used in the **`extends`** clause that cannot be expressed in this context-free grammar: Each **`AnnotationType`** mentioned in an **`extends`** clause must itself be an annotation type introduced using the **`AnnotationTypeDeclaration`**.

Listing A.6 shows some examples of allowed and disallowed uses of subtyping for annotations.

The Java Specification Request "Annotations on Types" (JSR 308) [47] listed one problem when annotations allow subtyping: There may be trust issues, because an annotation in a secure framework may be subclassed, and then a non-secure annotation may be used. I do not consider that a major issue: The problem with untrusted code exists for regular classes and frameworks as well. To ensure that an annotation cannot be subclassed, I allowed the `final` modifier for annotations, just as it is allowed for non-abstract classes. The example in listing A.7 below would generate an error.

Considering how small the changes were, I strongly urge Oracle to incorporate subtyping for annotations.

```
// java.lang.annotation.Annotation as implicit superclass
@interface BaseAnnotation {
  int value();
}

// extends BaseAnnotation
@interface SubAnnotation extends BaseAnnotation {
  String s();
}

// java.lang.annotation.Annotation as implicit superclass
@interface AnotherAnnotation {
  Class c();
}

// java.lang.annotation.Annotation as implicit superclass
@interface ThirdAnnotation {
  Class value();
}

// extends both SubAnnotation and AnotherAnnotation
@interface SubSubAnnotation extends SubAnnotation,
                                    AnotherAnnotation {
}

// error: Annotation is not an annotation itself
@interface ErroneousAnnotation
      extends java.lang.annotation.Annotation {
  int value();
}

// error: value member already exists in BaseAnnotation
// cannot be overridden
@interface ErroneousAnnotation2 extends BaseAnnotation {
  int value();
}

// error: value member already exists in BaseAnnotation
// cannot be overloaded based on return type
@interface ErroneousAnnotation3 extends BaseAnnotation {
  String value();
}

// error: value member exists in both BaseAnnotation
// and ThirdAnnotation --> ambiguous
@interface ErroneousAnnotation4 extends BaseAnnotation,
                                        ThirdAnnotation {
}
```

Listing A.6: Examples of Allowed and Disallowed Uses of Subtyping for Annotations

```
final @interface FinalAnnotation {                                    1
  String value();                                                     2
}                                                                     3
                                                                      4
// error: cannot inherit from final FinalAnnotation                   5
@interface SubAnnotation extends FinalAnnotation {                    6
  int i();                                                            7
}                                                                     8
```

Listing A.7: Using `final` to Prevent Extending Annotations

Appendix B

Sample Source Code

This appendix contains the source code to the examples that were used to evaluate the effectiveness of the Concutest framework. The source is being included so as to not repeat the problems that ConTest's [9] and rstest's [42] limited availability caused.

The source code of these samples, along with all other source code for Concutest, is also available at: `http://www.concutest.org/` [33].

B.1 Scheduling Experiment Source Code

B.1.1 Experiment 1: Race

```
public class ConTestOne {                                                    1
  private final static int NUM = 3;                                          2
  private static boolean first = true;                                       3
  private static final java.util.concurrent.atomic.AtomicInteger             4
    count = new java.util.concurrent.atomic.AtomicInteger(0);                5
                                                                             6
  private static class Racer extends Thread {                                7
    public void run() {                                                      8
      if (first) {                                                           9
        first = false;                                                       10
        System.out.println("first");                                         11
        count.incrementAndGet();                                             12
      }                                                                      13
    }                                                                        14
  }                                                                          15
                                                                             16
  public static void main(String[] args)                                     17
    throws InterruptedException {                                            18
    Racer[] racers = new Racer[NUM];                                         19
    for (int i=0; i<NUM; i++) {                                              20
```

```
      racers[i] = new Racer();
    }
    for (Racer r : racers) {
      r.start();
    }
    for (Racer r : racers) {
      r.join();
    }
    System.out.println("final count: " + count.get());
    if (count.get() != 1) {
      System.err.println("   Bug - expected 1");
    }
  }
}
```

Listing B.1: Several threads, each normally too short to be preempted, race to be the first thread to set a flag

B.1.2 Experiment 2: Atomicity

```
class Stocks implements FundConstants {
  static int[] balances = new int[noOfStocks];
  static { for (int n=0;n<noOfStocks;n++) balances[n] = 10000; }
  static void transfer(Transfer t) {
    balances[t.fundFrom] -= t.amount;
    balances[t.fundTo] += t.amount;
    BusyWork.doStuff();
  }
  static void checkSystem() {
    int actual = 0;
    for (int n=0;n<noOfStocks;n++) { actual += balances[n]; }
  }
}
class BusyWork {
  public static int[] dummy = new int[10000];
  public static int alwaysZero() {
    if (true) return 0; else return 1;
  }
  public static void doStuff() {
    java.util.Random r = new java.util.Random();
    for (int i=0; i<dummy.length; ++i) {
      dummy[i] = r.nextInt();
    }
  }
}
interface FundConstants {
  static final int noOfStocks = 3, noOfManagers = 150,
    testDuration = 20, randomTransfers = 1000,
```

```
    initialBalance = 10000,
    totalMoneyInUniverse = noOfStocks * initialBalance;
}
class Transfer implements FundConstants {
  final public int fundFrom, fundTo, amount;
  public Transfer() {
    fundFrom = (int)(Math.random() * noOfStocks);
    fundTo = (int)(Math.random() * noOfStocks);
    amount = (int)(Math.random() * 1000);
  }
}
class FundManager implements Runnable, FundConstants {
  public void run() {
    int next = 0;
    while (true) {
      Stocks.transfer(TestFundManagers.transfers[next++]);
      if (next == randomTransfers) next = 0;
      try { Thread.sleep(1); }
      catch (InterruptedException ie) { return; }
    }
  }
}
```

Listing B.2: Several threads race to read and write shared data, but the modifications are not atomic. This example was originally presented in a JDC Tech Tip [48]

```
public class TestFundManagers implements FundConstants {
  public static Transfer transfers[] =
    new Transfer[randomTransfers];
  static {
    for (int n=0; n<randomTransfers; n++) {
      transfers[n] = new Transfer();
    }
  }
  public static void main(String [] args) {
    Thread[] threads = new Thread[noOfManagers];
    FundManager[] mgrs = new FundManager[noOfManagers];
    for (int n=0; n<noOfManagers; n++) {
      mgrs[n] = new FundManager();
      threads[n] = new Thread(mgrs[n]);
      threads[n].setPriority(1 + (int)(Math.random() * 4));
      threads[n].start();
    }
    for (int n=0; n<testDuration; n++) {
      try {
        Thread.sleep(1000);
        Stocks.checkSystem();
      }
      catch (InterruptedException ie) {}
```

```
    }
    System.out.println();
    for (int n=0; n<noOfManagers; n++) {
      threads[n].interrupt();
    }
    for (int n=0; n<noOfManagers; n++) {
      try {
        threads[n].join();
      } catch (InterruptedException ie) {}
    }
    Stocks.checkSystem();
  }
}
```

Listing B.3: Test class for the code in listing B.2. This example was originally presented in a JDC Tech Tip [48]

B.1.3 Experiment 3: Uninitialized Data

```
public class ConTestThree {
  public static final int NUM_THREADS = 4;
  public static void main(String[] args) throws Exception {
    ChangeNotification[] cns =
      new ChangeNotification[NUM_THREADS];
    Thread[] ts = new Thread[NUM_THREADS];

    for(int i=0; i<NUM_THREADS; ++i) {
      cns[i] = new ChangeNotification();
      ts[i] = new Thread(cns[i]);
    };
    for(Thread t: ts) { t.start(); }

    Thread.sleep(1000);

    for(ChangeNotification cn: cns) {
      Subject s = new Subject();
      cn.changeNotification(s);
    }
    for(Thread t: ts) { t.join(); }
  }
}
class ChangeNotification implements Runnable {
  static boolean notified = false;
  public void run() {
    while(notified == false) {
      Thread.currentThread().yield();
    }
    System.out.println("subject current value is:"
```

```
                        + subject.currentValue());
  }
  public void changeNotification(Subject subject) {
    notified = true;
    this.subject = subject;
  }
  public Subject subject;
}
class Subject {
  public long currentValue() {
    return System.identityHashCode(this); }
}
```

Listing B.4: A main thread and several child threads have a data race on a flag, which may cause child threads to use uninitialized data

B.1.4 Experiment 4: Chain of Threads

```
import java.util.*;

public class ConTestFour {
  public static final int NUM_THREADS = 10;

  public static final HashMap<Integer,String> map =
    new HashMap<Integer,String>();

  public static void main(String[] args) throws Exception {
    ChildThread child = new ChildThread(1);
    child.start();
    map.put(1, "x");
    child.join();
  }


  public static class ChildThread extends Thread {
    int key;
    public ChildThread(int k) {
      super();
      key = k;
    }

    public void run() {
      String v = map.get(key);
      String v2 = v.toString();
      if (key<NUM_THREADS) {
        ChildThread child = new ChildThread(key+1);
        child.start();
        map.put(key+1, map.get(key)+"x");
```

```
        try {
          child.join();
        }
        catch(InterruptedException ie) {
        }
      }
    }
  }
}
```

Listing B.5: Threads are spawned recursively in a chain. If the child thread starts executing immediately, it may use a hash map key that does not exist.

B.1.5 Experiment 5: Missed Notification

```
public class RSTestOne {
  private static final int NUM = 3;
  private static final int ITERATIONS = 100;

  public static class Event {
    int count = 0;
    public Event() { }
    public Event(int c) { count = c; }
    public synchronized void wait_for_event() {
      try{ wait(); }
      catch(InterruptedException e) { }
    }
    public synchronized void signal_event(){
      count = count + 1;
      notifyAll();
    }
  }

  public static class Planner extends Thread {
    Event event1, event2;
    int count = 0;

    public Planner() {
      this(new Event(), new Event());
    }

    public Planner(Event e1, Event e2) {
      event1 = e1;
      event2 = e2;
      count = event1.count;
    }

    public void run(){
```

```
      int iterations = 0;
      while(iterations<ITERATIONS) {
        ++iterations;

        if (count == event1.count) {
          event1.wait_for_event();
        }
        count = event1.count;

        /* Generate plan */
        RSTestOne.sleep(1);

        event2.signal_event();
      }
    }
  }

  public static void main(String[] args)
    throws InterruptedException {
    sleep(5000);
    Event e1 = new Event(1);
    Event e2 = new Event();
    Planner p = new Planner(e1, e2);
    p.start();

    Thread t = new Thread() {
      public void run() {
        RSTestOne.sleep(5000);
        System.exit(0);
      }
    };
    t.setDaemon(true);
    t.start();

    int iterations = 0;
    while(p.isAlive() && (iterations<ITERATIONS)) {
      e1.signal_event();
      e2.wait_for_event();
    }
  }

  public static void sleep(int ms) {
    try { Thread.sleep(ms); }
    catch(InterruptedException ie) { }
  }
}
```

Listing B.6: Thread in NASA's Remote Agent may miss `notifyAll` and wait forever.